$P4_{16}$ Portable Switch Architecture (PSA)

(working draft)

The P4.org Architecture Working Group

2017-12-22

Abstract

P4 is a language for expressing how packets are processed by the data plane of a programmable network forwarding element. P4 programs specify how the various programmable blocks of a target architecture are programmed and connected. The Portable Switch Architecture (PSA) is target architecture that describes common capabilities of network switch devices which process and forward packets across multiple interface ports.

Contents

1. Target Architecture Model

The Portable Switch Architecture (PSA) Model has six programmable P4 blocks and two fixed-function blocks, as shown in Figure 1. Programmable blocks are hardware blocks whose function can be programmed using the P4 language. The Packet buffer and Replication Engine (PRE) and the Buffer Queuing Engine (BQE) are target dependent functional blocks that may be configured for a fixed set of operations.

Incoming packets are parsed and validated, and then passed to an ingress match action pipeline, which makes decisions on where the packets go. After the ingress pipeline, the packet may be buffered and/or replicated (sent to multiple egress ports). For each such egress port, the packet passes through an egress match action pipeline before it is deparsed and queued to leave the pipeline. Note: the checksum operations are available – validation in the parser, and checksum computation and update in deparser (see also Table 5).

A programmer targeting the PSA is required to instantiate objects for the programmable blocks that conform to these APIs (see section 5). Note that the programmable block APIs are templatized on user defined headers and metadata. In PSA, the user can define a single metadata type for all controls.

When instantiating the `main` `package` object, the instances corresponding to the programmable blocks are passed as arguments.

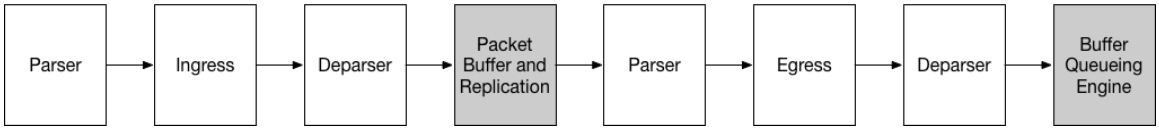

Figure 1. Portable Switch Pipeline

A P4 programmer wishing to maximize the portability of their program should follow several general guidelines:

- Do not use undefined values in a way that affects the resulting output packet(s), or for side effects such as updating `Counter`, `Meter` or `Register` instances.
- Use as few resources as possible, e.g. table search key bits, array sizes, quantity of metadata associated with packets, etc.

2. Naming conventions

In this document we use the following naming conventions:

- Types are named using CamelCase followed by `_t`. For example, `PortId_t`.
- Control types and extern object types are named using CamelCase. For example `IngressParser`.
- Struct types are named using lower case words separated by `_` followed by `_t`. For example `psa_ingress_input_metadata_t`.
- Actions, extern methods, extern functions, headers, structs, and instances of controls and externs start with lower case and words are separated using `_`. For example `send_to_port`.
- Enum members, const definitions, and #define constants are all caps, with words separated by `_`. For example `PORT_CPU`.

Architecture specific metadata (e.g. structs) are prefixed by `psa_`.

3. Packet paths

Figure 2 shows all possible paths for packets that must be supported by a PSA implementation. An implementation is allowed to support paths for packets that are not described here.

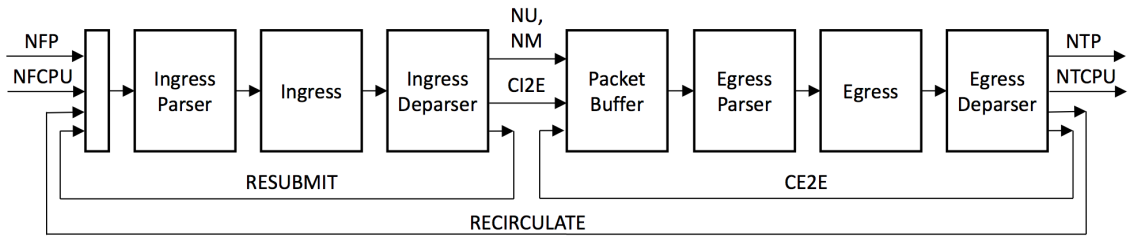

Figure 2. Packet Paths in PSA

Table 1 defines the meanings of the abbreviations in Figure 2. There can be one or more hardware, software, or PSA architecture components between the "packet source" and "packet destination" given in that table, e.g. a normal multicast packet passes through the packet replication engine and typically also a packet buffer after leaving the ingress deparser, before arriving at the egress parser.

Abbreviation	Description	Packet Source	Packet Destination
NFP	normal packet from port	port	ingress parser
NFCPU	packet from CPU port	CPU port	ingress parser
NU	normal unicast packet from ingress to egress	ingress deparser	egress parser
NM	normal multicast-replicated packet from ingress to egress	ingress deparser, with help from PRE	egress parser (more than one copy is possible)
NTP	normal packet to port	egress deparser	port
NTCPU	normal packet to CPU port	egress deparser	CPU port
RESUB	resubmitted packet	ingress deparser	ingress parser
CI2E	clone from ingress to egress	ingress deparser	egress parser
RECIRC	recirculated packet	egress deparser	ingress parser
CE2E	clone from egress to egress	egress deparser	egress parser

Table 1. Packet path abbreviation meanings.

Abbreviation	Description	Processed next by	Resulting packet(s)
NFP	normal packet from port	ingress	At most one CI2E packet, plus at most one of a RESUB, NU, or NM packet. See section 6.2 for details.
NFCPU	packet from CPU port		
RESUB	resubmitted packet		
RECIRC	recirculated packet		
NU	normal unicast packet from ingress to egress	egress	At most one CE2E packet, plus at most one of a RECIRC, NTP, or NTCPU packet. See section 6.5 for details.
NM	normal multicast-replicated packet from ingress to egress		
CI2E	clone from ingress to egress		
CE2E	clone from egress to egress		
NTP	normal packet to port	device at other end of port	determined by the other device
NTCPU	normal packet to CPU port	CPU	determined by CPU

Table 2. Result of packet processed one time by ingress or egress.

This table focuses on the P4-programmable portions of the architecture as sources and destinations of these packet paths.

Table 2 shows what can happen to a packet as a result of a single time being processed in ingress, or a single time being processed in egress. The cases are the same as shown in Table 1, but have been grouped together by similar values of "Processed next by".

There are metadata fields defined by PSA that enable your P4 program to identify which path each packet arrived on, and to control where it will go next. See section 6.

For egress packets, the choice between one of multiple egress ports, the port to the CPU, or the "recirculation port", is made by the immediately previous processing step (ingress for NU, NM, or CI2E packets, egress for CE2E packets). Egress processing can choose to drop the packet instead of sending it to the port chosen earlier, but it cannot change the choice to a different port. Ingress code is the most common place in a P4 program where the output port(s) are chosen. The only exception to ingress choosing the output port is for egress-to-egress clone packets, whose destination port is chosen when the clone is created in the immediately preceding egress processing step. See section D.2 for why this restriction exists.

A single packet received by a PSA system from a port can result in 0, 1, or many packets going

out, all under control of the P4 program. For example, a single packet received from a port could cause all of the following to occur, if the P4 program so directed it:

- The original packet received as NFP from port 2. Ingress processing creates a CI2E clone destined for the CPU port (copy 1), and a multicast NM packet to multicast group 18, which is configured in the PacketReplicationEngine to have copies made to ports 5 (copy 2) and the recirculate port `PORT_RECIRCULATE` (copy 3).
- Copy 1 performs egress processing, which sends the packet on path NTCPU to the CPU port.
- Copy 2 performs egress processing, which creates a CE2E clone destined for port 8 (copy 4), and sends a NTP packet to port 5.
- Copy 3 performs egress processing, which does a RECIRC back to ingress (copy 5).
- Copy 4 performs egress processing, which sends a NTP packet to port 8.
- Copy 5 performs ingress processing, which sends a NU packet destined for port 1 (copy 6).
- Copy 6 performs egress processing, which drops the packet instead of sending it to port 1.

This is simply an example of what is possible given an appropriately written P4 program. There is no need to use all of the packet paths available. The numbering of the packet copies above is only for purposes of distinctly identifying each one in the example. The ports described in the example are also arbitrary. A PSA implementation is free to perform the steps above in many possible orders.

There is no mandated mechanism in PSA to prevent a single received packet from creating packets that continue to recirculate, resubmit, or clone from egress to egress indefinitely. This can be prevented by suitable testing of your P4 program, and/or creating in your P4 program a "time to live" metadata field that is carried along with copies of a packet, similar to the IPv4 Time To Live header field.

A PSA implementation may optionally drop resubmitted, recirculated, or egress-to-egress clone packets after an implementation-specific maximum number of times from the same original packet. If so, the implementation should maintain counters of packets dropped for these reasons, and preferably record some debug information about the first few packets dropped for these reasons (perhaps only one).

4. PSA Data types

4.1. PSA type definitions

Each PSA implementation will have specific bit widths for the following types. These widths should be defined in the target specific implementation of the PSA file.

The P4 Runtime API plans to use 64-bit values to hold values of these types. Typical PSA implementations are expected to choose significantly smaller sizes than 64 bits for the data plane implementation of these types. If for some reason a PSA implementation chooses to use more than 64 bits for one or more of these types, note that the P4 Runtime API will be able to take advantage of at most 64 of those bits.

```
typedef bit<unspecified> PortId_t;
typedef bit<unspecified> MulticastGroup_t;
typedef bit<unspecified> CloneSessionId_t;
typedef bit<unspecified> ClassOfService_t;
typedef bit<unspecified> PacketLength_t;
typedef bit<unspecified> EgressInstance_t;
typedef bit<unspecified> Timestamp_t;

const   PortId_t         PORT_RECIRCULATE = unspecified;
const   PortId_t         PORT_CPU = unspecified;

const   CloneSessionId_t PSA_CLONE_SESSION_TO_CPU = unspecified;
```

4.2. PSA supported metadata types

```
enum PacketPath_t {
    NORMAL,     /// Packet received by ingress that is none of the cases below.
    NORMAL_UNICAST,   /// Normal packet received by egress which is unicast
    NORMAL_MULTICAST, /// Normal packet received by egress which is multicast
    CLONE_I2E,  /// Packet created via a clone operation in ingress,
                /// destined for egress
    CLONE_E2E,  /// Packet created via a clone operation in egress,
                /// destined for egress
    RESUBMIT,   /// Packet arrival is the result of a resubmit operation
    RECIRCULATE /// Packet arrival is the result of a recirculate operation
}

struct psa_ingress_parser_input_metadata_t {
  PortId_t                ingress_port;
  PacketPath_t            packet_path;
}

struct psa_egress_parser_input_metadata_t {
  PortId_t                egress_port;
  PacketPath_t            packet_path;
}

struct psa_ingress_input_metadata_t {
  // All of these values are initialized by the architecture before
  // the Ingress control block begins executing.
  PortId_t                ingress_port;
  PacketPath_t            packet_path;
  Timestamp_t             ingress_timestamp;
  ParserError_t           parser_error;
}

struct psa_ingress_output_metadata_t {
  // The comment after each field specifies its initial value when the
  // Ingress control block begins executing.
  ClassOfService_t        class_of_service; // 0
  bool                    clone;            // false
  CloneSessionId_t        clone_session_id; // initial value is undefined
  bool                    drop;             // true
  bool                    resubmit;         // false
  MulticastGroup_t        multicast_group;  // 0
  PortId_t                egress_port;      // initial value is undefined
}

struct psa_egress_input_metadata_t {
  ClassOfService_t        class_of_service;
  PortId_t                egress_port;
  PacketPath_t            packet_path;
  EgressInstance_t        instance;       /// instance comes from the PacketReplicationEngine
  Timestamp_t             egress_timestamp;
  ParserError_t           parser_error;
```

```
}

/// This struct is an 'in' parameter to the egress deparser.  It
/// includes enough data for the egress deparser to distinguish
/// whether the packet should be recirculated or not.
struct psa_egress_deparser_input_metadata_t {
  PortId_t                egress_port;
}

struct psa_egress_output_metadata_t {
  // The comment after each field specifies its initial value when the
  // Egress control block begins executing.
  bool                    clone;         // false
  CloneSessionId_t        clone_session_id; // initial value is undefined
  bool                    drop;          // false
}
```

4.3. Match kinds

Additional supported match_kind types

```
match_kind {
    range,   /// Used to represent min..max intervals
    selector /// Used for implementing dynamic_action_selection
}
```

5. Programmable blocks

The following declarations provide a template for the programmable blocks in the PSA. The P4 programmer is responsible for implementing controls that match these interfaces and instantiate them in a package definition.

It uses the same user-defined metadata type `IM` and header type `IH` for all ingress parsers and control blocks. The egress parser and control blocks can use the same types for those things, or different types, as the P4 program author wishes.

```
parser IngressParser<H, M, RESUBM, RECIRCM>(
    packet_in buffer,
    out H parsed_hdr,
    inout M user_meta,
    in psa_ingress_parser_input_metadata_t istd,
    in RESUBM resubmit_meta,
    in RECIRCM recirculate_meta);

control Ingress<H, M>(
    inout H hdr, inout M user_meta,
    in    psa_ingress_input_metadata_t  istd,
    inout psa_ingress_output_metadata_t ostd);

control IngressDeparser<H, M, CI2EM, RESUBM, NM>(
    packet_out buffer,
    out CI2EM clone_i2e_meta,
    out RESUBM resubmit_meta,
```

```
    out NM normal_meta,
    inout H hdr,
    in M meta,
    in psa_ingress_output_metadata_t istd);

parser EgressParser<H, M, NM, CI2EM, CE2EM>(
    packet_in buffer,
    out H parsed_hdr,
    inout M user_meta,
    in psa_egress_parser_input_metadata_t istd,
    in NM normal_meta,
    in CI2EM clone_i2e_meta,
    in CE2EM clone_e2e_meta);

control Egress<H, M>(
    inout H hdr, inout M user_meta,
    in    psa_egress_input_metadata_t  istd,
    inout psa_egress_output_metadata_t ostd);

control EgressDeparser<H, M, CE2EM, RECIRCM>(
    packet_out buffer,
    out CE2EM clone_e2e_meta,
    out RECIRCM recirculate_meta,
    inout H hdr,
    in M meta,
    in psa_egress_output_metadata_t istd,
    in psa_egress_deparser_input_metadata_t edstd);

package IngressPipeline<IH, IM, NM, CI2EM, RESUBM, RECIRCM>(
    IngressParser<IH, IM, RESUBM, RECIRCM> ip,
    Ingress<IH, IM> ig,
    IngressDeparser<IH, IM, CI2EM, RESUBM, NM> id);

package EgressPipeline<EH, EM, NM, CI2EM, CE2EM, RECIRCM>(
    EgressParser<EH, EM, NM, CI2EM, CE2EM> ep,
    Egress<EH, EM> eg,
    EgressDeparser<EH, EM, CE2EM, RECIRCM> ed);

package PSA_Switch<IH, IM, EH, EM, NM, CI2EM, CE2EM, RESUBM, RECIRCM> (
    IngressPipeline<IH, IM, NM, CI2EM, RESUBM, RECIRCM> ingress,
    PacketReplicationEngine pre,
    EgressPipeline<EH, EM, NM, CI2EM, CE2EM, RECIRCM> egress,
    BufferingQueueingEngine bqe);
```

6. Packet Path Details

Refer to section 3 for the packet paths provided by PSA, and their abbreviated names, used often in this section.

	NFP	NFCPU	RESUB	RECIRC
packet_in	see text			
user_meta	see text			
IngressParser istd fields (type psa_ingress_parser_input_metadata_t)				
ingress_port	PortId_t value of packet's input port	PORT_CPU	copied from resub'd packet	PORT_RECIRCULATE
packet_path	NORMAL	NORMAL	RESUBMIT	RECIRCULATE
Ingress istd fields (type psa_ingress_input_metadata_t)				
ingress_port	Same value as received by IngressParser above.			
packet_path	Same value as received by IngressParser above.			
ingress_timestamp	Time that packet began processing in IngressParser. For RESUB or RECIRC packets, the time the 'copy' began IngressParser, not the original.			
parser_error	From IngressParser. Always error.NoError if there was no parser error.			

Table 3. Initial values for packets processed by ingress.

6.1. Initial values of packets processed by ingress

Table 3 describes the initial values of the packet contents and metadata when a packet begins ingress processing.

Note that the ingress_port value for a resubmitted packet could be PORT_RECIRCULATE if a packet was recirculated, and then that recirculated packet was resubmitted.

6.1.1. Initial packet contents for packets from ports

For Ethernet ports, packet_in for FP and NFCPU path packets contains the Ethernet frame starting with the Ethernet header. It does not include the Ethernet frame CRC.
TBD: Whether the payload is always the minimum of 46 bytes (64 byte minimum Ethernet frame size, minus 14 bytes of header, minus 4 bytes of CRC), or whether an implementation is allowed to leave some of those bytes out.}
The PSA does not put further restrictions on packet_in.length() as defined in the $P4_{16}$ spec. Targets that do not support it, should provide mechanisms to raise an error.

6.1.2. Initial packet contents for resubmitted packets

For RESUB packets, packet_in is the same as the pre-IngressParser contents of packet_in, for the packet that caused this resubmitted packet to occur (i.e. with NO modifications from any ingress processing).

6.1.3. Initial packet contents for recirculated packets

For RECIRC packets, packet_in is created by starting with the headers emitted by the egress deparser of the egress packet that was recirculated, followed by the payload of that packet, i.e. the part that was not parsed by the egress parser.

6.1.4. User-defined metadata for all ingress packets

The PSA architecture does not mandate initialization of user-defined metadata to known values as given as input to the ingress parser. If a user's P4 program explicitly initializes all user-defined metadata early on (e.g. in the parser's start state), then that will flow through the rest of the

parser into the `Ingress` control block as one might normally expect. This will be left as an option to the user in their P4 programs, not required behavior for all P4 programs.

There are two direction `in` parameters to the ingress parser with user-defined types, named `resubmit_meta` and `recirculate_meta`. They may be used to carry metadata for resubmitted and recirculated packets.

Consider a packet that arrives at the ingress pipeline, and during ingress processing the P4 program assigns values to fields of the PSA standard metadata such that the packet is resubmitted (see Section 6.2 for details on how to do so). In the ingress deparser, the P4 program assigns a value to the `out` parameter named `resubmit_meta`. This value (which can be a collection of many individual values in fields, sub-structs, headers, etc.) becomes associated with the resubmitted packet by the PSA implementation, and when the resubmitted packet begins ingress parsing, that becomes the value of the `in` parameter named `resubmit_meta` to the ingress parser.

For resubmitted packets, the value of the `in` parameter named `recirculate_meta` is undefined.

Conversely, for recirculated packets, the value of the `in` parameter named `recirculate_meta` contains whatever value was assigned to the egress deparser `out` parameter named `recirculate_meta` when the packet was recirculated. The value of the `in` parameter `resubmit_meta` is undefined for recirculated packets.

For packets from a port, including the CPU port, both of the `in` parameters `resubmit_meta` and `recirculate_meta` are undefined.

6.2. Behavior of packets after ingress processing is complete

The pseudocode below defines where copies of packets will be made after the `Ingress` control block has completed executing, based upon the contents of several metadata fields in the struct `psa_ingress_output_metadata_t`.

The function `platform_port_valid()` mentioned below takes a value of type `PortId_t`, returning `true` only when the value represents an output port for the implementation. It is expected that for some PSA implementations there will be bit patterns for a value of type `PortId_t` that do not correspond to any port. This function returns true for both `PORT_CPU` and `PORT_RECIRCULATE`. `platform_port_valid` is not defined in PSA for calling from the P4 data-plane program, since there is no known use case for calling it at packet processing time. It is intended for describing the behavior in pseudo-code. The control plane is expected to configure tables with valid port numbers.

A comment saying "recommended to log error" is not a requirement, but a recommendation, that a PSA implementation should maintain a counter that counts the number of times this error occurs. It would also be useful if the implementation recorded details about the first few times this error occurred, e.g. a FIFO queue of the first several invalid values of `ostd.egress_port` that cause an error to occur, perhaps with other information about the packet that caused it, with tail dropping if it fills up. Control plane or driver software would be able to read these counters, and read and drain the FIFO queues to assist P4 developers in debugging their code.

```
struct psa_ingress_output_metadata_t {
  // The comment after each field specifies its initial value when the
  // Ingress control block begins executing.
  ClassOfService_t         class_of_service; // 0
  bool                     clone;            // false
  CloneSessionId_t         clone_session_id; // initial value is undefined
  bool                     drop;             // true
  bool                     resubmit;         // false
  MulticastGroup_t         multicast_group;  // 0
  PortId_t                 egress_port;      // initial value is undefined
}

    psa_ingress_output_metadata_t ostd;
```

```
if (ostd.clone) {
    if (ostd.clone_session_id value is supported) {
        cos = class_of_service configured for ostd.clone_session_id in PRE
        ep = egress_port configured for ostd.clone_session_id in PRE
        trunc = truncate configured for ostd.clone_session in PRE
        plen = packet_length_bytes configured for ostd.clone_session in PRE
        if (cos value is not supported) {
            cos = 0;
            // Recommmended to log error about unsupported cos
            // value.
        }
        if (platform_port_valid(ep)) {
            create a clone of the packet and send it to the packet
                buffer with the egress_port ep and
                class_of_service cos, after which it will start
                egress processing.  It will contain at most the
                first plen bytes of the packet as received at the
                ingress parser if trunc is true, otherwise the
                entire packet.
        } else {
            // Do not create a clone.  Recommended to log error
            // about unsupported ep value.
        }
    } else {
        // Do not create a clone.  Recommmended to log error about
        // unsupported ostd.clone_session_id value.
    }
}
// Continue below, regardless of whether a clone was created.
// Any clone created above is unaffected by the code below.
if (ostd.drop) {
    drop the packet
    return;   // Do not continue below.
}
if (ostd.class_of_service value is not supported) {
    ostd.class_of_service = 0;    // use default class 0 instead
    // Recommended to log error about unsupported
    // ostd.class_of_service value.
}
if (ostd.resubmit) {
    resubmit the packet, i.e. it will go back to starting with the
        ingress parser;
    return;   // Do not continue below.
}
if (ostd.multicast_group != 0) {
    Make 0 or more copies of the packet according to the control
        plane configuration of multicast group ostd.multicast_group.
        Every copy will have the same value of ostd.class_of_service
    return;   // Do not continue below.
}
if (platform_port_valid(ostd.egress_port)) {
```

```
        enqueue one packet for output port ostd.egress_port with class
            of service ostd.class_of_service
    } else {
        drop the packet
        // Recommended to log error about unsupported ostd.egress_port
        // value.
    }
```

Whenever the pseudocode above indicates that a packet should be sent on a particular packet path, a PSA implementation may under some circumstances instead drop the packet. For example, the packet buffer may be too low on available space for storing new packets, or some other congestion control mechanism such as RED (Random Early Detection) or AFD (Approximate Fair Dropping) may select the packet for dropping. It is recommended that an implementation maintain counters of packets dropped, preferably with separate counters for as many different reasons as the implementation has for dropping packets outside the control of the P4 program.

A PSA implementation may implement multiple classes of service for packets sent to the packet buffer. If so, the `Ingress` control block may choose to assign a value to the `ostd.class_of_service` field to change the packet's class of service to a value other than the default of 0.

PSA only specifies how the `Ingress` control block can control the class of service of packets. PSA does not mandate a scheduling policy among queues that may exist in the packet buffer. Something at least as flexible as weighted fair queuing, with an optional strict high priority queue, is recommended for PSA implementations with separate queues for each class of service.

Normally all unicast packets (i.e. those that follow the "enqueue one packet" path in the pseudocode above) received by a PSA device on the same ingress port, and sent to the same output port, will be processed by the `Ingress` control block in the same order they are received, and then processed by the `Egress` control bock in the same relative order as they are processed by the `Ingress` control block, i.e. all such packets go through the same FIFO queue in the packet buffer.

It is expected that some PSA implementations will implement the class of service mechanism by having a separate FIFO queue per class of service, and thus while unicast packets with the same ingress port, egress port, and class of service will pass through the system in FIFO order, unicast packets with the same ingress and egress port, but different classes of service, may be processed by the `Egress` control block in a different order than they were processed by the Ingress control block.

A similar FIFO property is expected to hold for multicast packets, i.e. those that follow the "Make 0 or more copies" path in the pseudocode above. By a short period of time after the control plane has stopped modifying the set of members of multicast groups, they should be 'stable', and all packets with the same (`ingress_port`, `ostd.multicast_group`, `ostd.class_of_service`, `egress_port`) should pass through the system in FIFO order, and be processed in the `Egress` control block in the same relative order that they were processed in the `Ingress` control block.

There is no such expectation for multicast packets with different `class_of_service` values, again because of separate queues in the PRE for different `class_of_service` values.

The control plane API excerpt below is an example intended to be added as part of the P4 Runtime API, and the final version of this message will be defined by the P4 Runtime WG.

```
// The ClassOfServiceInfo message should be added to the "oneof"
// inside of message "Entity".

// ClassOfServiceInfo is only intended to be read.  Attempts to update
// this entity have no effect, and should return an error status that
// the entity is read only.

message ClassOfServiceInfo {
  // The number of class of service queues per output port that are
  // available in this PSA implementation.
  uint32 class_of_service_queues_per_output_port = 1;
```

```
  // The list of values of type ClassOfService_t that are supported by
  // this PSA implementation.  It is recommended that they be a
  // contiguous range from 0 up to
  // (class_of_service_queues_per_output_port - 1).
  repeated uint32 class_of_service_id = 2;
}
```

6.2.1. Multicast replication

The control plane may configure each `multicast_group` in the PRE to create the desired copies of packets sent to that group. Each group begins empty. Sending a packet to an empty group causes the packet to be dropped. The control plane may add one or more pairs of the form `(egress_port, instance)` to a multicast group, and may also remove pairs from a group that were added earlier.

Suppose a multicast group contains the following set of pairs:

```
(egress_port[0], instance[0]),
(egress_port[1], instance[1]),
...,
(egress_port[N-1], instance[N-1])
```

When a packet is sent to that group, `N` copies of the packet are made. Copy number `i` that is sent to egress processing will have its `struct` of type `psa_egress_input_metadata_t` filled in with the field `egress_port` equal to `egress_port[i]`, and the field `instance` filled in with `instance[i]`. Note: A multicast group is a set of pairs, and it is not required that an implementation create copies in an order that the control plane can enforce. However, see the ordering constraints for multicast described in 6.2.

Within a single multicast group, the pairs `(egress_port, instance)` must be different from each other, but it is allowed for any number of pairs within a multicast group to have the same value of `egress_port`, or to have the same value of `instance`. The same pair `(egress_port, instance)` can occur in any number of different multicast groups.

A PSA implementation need only support `egress_port` values that represent single ports of the PSA device. That is, it need not implement support for `egress_port` values that represent an entire Link Aggregation Group (LAG) interface, which is a set of physical ports over which load balancing of traffic is performed.

A PSA device must support `egress_port` values in a multicast group that are equal to `PORT_CPU` or `PORT_RECIRCULATE`. The copies of a multicast packet made to those ports will behave the same in egress as a unicast packets sent to the corresponding port, i.e. if not dropped, those copies will go the the CPU port, or be recirculated back to ingress.

6.3. Actions for directing packets during ingress

All of these actions modify one or more metadata fields in the struct with type `psa_ingress_output_metadata_t` that is an `out` parameter of the `Ingress` control block. None of these actions have any other immediate effect. What happens to the packet is determined by the value of all fields in that struct when ingress processing is complete, not at the time one of these actions is called. See Section 6.2.

These actions are provided for convenience in making changes to these metadata fields. Their effects are expected to be common kinds of changes one will want to make in a P4 program. If they do not suit your use cases, you may modify the metadata fields directly in your P4 programs however you prefer, e.g. within actions you define.

6.3.1. Unicast operation

Sends packet to a port. See Table 4, column NU, for how metadata fields are filled in when such a packet begins egress processing.

```
/// Modify ingress output metadata to cause one packet to be sent to
/// egress processing, and then to the output port egress_port.
/// (Egress processing may choose to drop the packet instead.)

/// This action does not change whether a clone or resubmit operation
/// will occur.

action send_to_port(inout psa_ingress_output_metadata_t meta,
                    in PortId_t egress_port)
{
    meta.drop = false;
    meta.multicast_group = 0;
    meta.egress_port = egress_port;
}
```

6.3.2. Multicast operation

Sends packet to a multicast group or a port. See Table 4, column NM, for how metadata fields are filled in when each multicast-replicated copy of such a packet begins egress processing.

The `multicast_group` parameter is the multicast group id. The control plane must configure the multicast groups through a separate mechanism such as the P4 Runtime API.

```
/// Modify ingress output metadata to cause 0 or more copies of the
/// packet to be sent to egress processing.

/// This action does not change whether a clone or resubmit operation
/// will occur.

action multicast(inout psa_ingress_output_metadata_t meta,
                 in MulticastGroup_t multicast_group)
{
    meta.drop = false;
    meta.multicast_group = multicast_group;
}
```

6.3.3. Drop operation

Do not send a copy of the packet for normal egress processing.

```
/// Modify ingress output metadata to cause no packet to be sent for
/// normal egress processing.

/// This action does not change whether a clone will occur.  It will
/// prevent a packet from being resubmitted.

action ingress_drop(inout psa_ingress_output_metadata_t meta)
{
    meta.drop = true;
}
```

	NU	NM	CI2E	CE2E
`packet_in`	see text			
`user_meta`	see text			
EgressParser `istd` fields (type `psa_egress_parser_input_metadata_t`)				
`egress_port`	`ostd.egress_port` of ingress packet	from PRE configuration of multicast group	from PRE configuration of clone session	
`packet_path`	NORMAL_ UNICAST	NORMAL_ MULTICAST	CLONE_I2E	CLONE_E2E
Egress `istd` fields (type `psa_egress_input_metadata_t`)				
`class_of_service`	`ostd.class_of_service` of ingress packet		from PRE configuration of clone session	
`egress_port`	Same value as received by EgressParser above.			
`packet_path`	Same value as received by EgressParser above.			
`instance`	From PacketReplicationEngine configuration for NM packets. 0 for all other kinds of packets.			
`egress_timestamp`	Time that packet began processing in EgressParser. Filled in independently for each copy of a multicast-replicated packet.			
`parser_error`	From EgressParser. Always `error.NoError` if there was no parser error. See "Multicast copies" section.			

Table 4. Initial values for packets processed by egress.

6.4. Initial values of packets processed by egress

Table 4 describes the initial values of the packet contents and metadata when a packet begins egress processing.

6.4.1. Initial packet contents for normal packets

For NU and NM packets, `packet_in` comes from the ingress packet that caused this packet to be sent to egress. It starts with the packet headers as emitted by the ingress deparser, followed by the payload of that packet, i.e. the part that was not parsed by the ingress parser.

Packets to be recirculated, i.e. those sent to port `PORT_RECIRCULATE` via the normal unicast or multicast packet paths, fit into this category. They are not treated differently by a PSA implementation from normal unicast or multicast packets until they reach the egress deparser.

6.4.2. Initial packet contents for packets cloned from ingress to egress

For CI2E packets, `packet_in` is from the ingress packet that caused this clone to be created. It is the same as the pre-IngressParser contents of `packet_in` of that ingress packet, with no modifications from any ingress processing. Truncation of the payload is supported.

Packets cloned in ingress using a clone session configured with `clone_port` equal to `PORT_RECIRCULATE` fit into this category.

6.4.3. Initial packet contents for packets cloned from egress to egress

For CE2E packets, `packet_in` is from the egress packet that caused this clone to be created. It starts with the headers emitted by the egress deparser, followed by the payload of that packet, i.e. the part that was not parsed by the egress parser. Truncation of the payload is supported.

Packets cloned in egress using a clone session configured with `clone_port` equal to `PORT_RECIRCULATE` fit into this category.

6.4.4. User-defined metadata for all egress packets

This is very similar to how metadata is initialized for ingress packets. See Section 6.1.4.

The primary differences for egress packets are the different packet paths involved. There are three parameters with direction `in` for the egress parser, named `normal_meta`, `clone_i2e_meta`, and `clone_e2e_meta`. For every packet that begins egress processing, exactly one of those three has defined contents, and the other two have undefined contents.

For NU and NM packets, the parameter `normal_meta` is the only one with defined contents. Its value is the one that was assigned to the `out` parameter with the same name of the ingress deparser, when the normal packet was completing its ingress processing.

For CLONE_I2E packets, the parameter `clone_i2e_meta` is the only one with defined contents. Its value is the one that was assigned to the `out` parameter with the same name of the ingress deparser, when the clone was created.

For CLONE_E2E packets, the parameter `clone_e2e_meta` is the only one with defined contents. Its value is the one that was assigned to the `out` parameter with the same name of the egress deparser, when the clone was created.

6.4.5. Multicast copies

The following fields may differ among copies of a multicast-replicated packet that are processed in egress.

- `egress_port` - This field will typically differ among copies of a multicast-replicated packet, but it may also be the same for arbitrary copies, as determined by the control plane configuration of the PacketReplicationEngine. It is expected that the control plane will configure the PacketReplicationEngine so that each copy of the same original packet is assigned a unique value of the pair `(egress_port, instance)`.
- `instance` - See `egress_port`
- `egress_timestamp` - This value is filled in independently for each copy of a multicast-replicated packet. Depending upon the quantity of traffic destined to each output port, the timestamp could vary significantly between copies of the same original packet.
- `parser_error` - In the common case, this will typically be the same for every copy of the same original multicast-replicated packet. However, it is determined by the EgressParser P4 code for each copy independently, so if that parsing behavior depends upon a field that can differ among copies, e.g. `egress_port`, then `parser_error` can differ among copies.

All contents of a packet and its associated metadata, other than those mentioned above, will be the same for every copy of the same original multicast-replicated packet.

6.5. Behavior of packets after egress processing is complete

The pseudocode below defines where copies of packets will be made after the `Egress` control block has completed executing, based upon the contents of several metadata fields in the struct `psa_egress_output_metadata_t`.

```
struct psa_egress_output_metadata_t {
  // The comment after each field specifies its initial value when the
  // Egress control block begins executing.
  bool                     clone;         // false
  CloneSessionId_t         clone_session_id; // initial value is undefined
  bool                     drop;          // false
}

    psa_egress_input_metadata_t  istd;
    psa_egress_output_metadata_t ostd;
```

```
if (ostd.clone) {
    if (ostd.clone_session_id value is supported) {
        cos = class_of_service configured for ostd.clone_session_id in PRE
        ep = egress_port configured for ostd.clone_session_id in PRE
        trunc = truncate configured for ostd.clone_session in PRE
        plen = packet_length_bytes configured for ostd.clone_session in PRE
        if (cos value is not supported) {
            cos = 0;
            // Recommmended to log error about unsupported cos
            // value.
        }
        if (platform_port_valid(ep)) {
            create a clone of the packet and send it to the packet
                buffer with the egress_port ep and
                class_of_service cos, after which it will start
                egress processing.  It will contain at most the
                first plen bytes of the packet as sent out from
                the egress deparser if trunc is true, otherwise
                the entire packet.
        } else {
            // Do not create a clone.  Recommended to log error
            // about unsupported ep value.
        }
    } else {
        // Do not create a clone.  Recommmended to log error about
        // unsupported ostd.clone_session_id value.
    }
}
// Continue below, regardless of whether a clone was created.
// Any clone created above is unaffected by the code below.
if (ostd.drop) {
    drop the packet
    return;   // Do not continue below.
}
// The value istd.egress_port below is the same one that the
// packet began its egress processing with, as decided during
// ingress processing for this packet.  The egress code is not
// allowed to change it.
if (istd.egress_port == PORT_RECIRCULATE) {
    recirculate the packet, i.e. it will go back to starting with the
        ingress parser;
    return;   // Do not continue below.
}
enqueue one packet for output port istd.egress_port
```

As for the handling of a packet after ingress processing, a PSA implementation may drop a packet after egress processing, even if the pseudocode above says that a packet will be sent. For example, you may attempt to clone a packet after egress when the packet buffer is too full, or you may attempt to recirculate a packet when the ingress pipeline is busy handling other packets. It is recommended that an implementation maintain counters of packets dropped, preferably with separate counters for as many different reasons as the implementation has for dropping packets outside the control of the P4 program.

6.6. Actions for directing packets during egress

6.6.1. Drop operation

Do not send the packet out of the device after egress processing is complete.

```
/// Modify egress output metadata to cause no packet to be sent out of
/// the device.

/// This action does not change whether a clone will occur.

action egress_drop(inout psa_egress_output_metadata_t meta)
{
    meta.drop = true;
}
```

6.7. Contents of packets sent out to ports

There is no metadata associated with NTP and NTCPU packets.

They begin with the series of bytes emitted by the egress deparser. Following that is the payload, which are those packet bytes that were not parsed in the egress parser.

For Ethernet ports, any padding required to get the packet up to the minimum frame size required is done by the implementation, as well as calculation of and appending the Ethernet frame CRC.

It is expected that typical P4 programs will have explicit checks to avoid sending packets larger than a port's maximum frame size. A typical implementation will drop frames larger than this maximum supported size. It is recommended that they maintain error counters for such dropped frames.

6.8. Packet Cloning

Packet cloning is a mechanism to send a copy of a packet to a specified port, in addition to the 'regular' packet,

One use case for cloning is packet mirroring, i.e. send the packet to its normal destination according to other features implemented by the P4 program, and in addition, send a copy of the packet as received to another output port, e.g. to a monitoring device.

Packet cloning happens at the end of the ingress and/or egress pipeline. PSA specifies the following semantics for the clone operation. When the clone operation is invoked at the end of the ingress pipeline, the cloned packet is a copy of the packet as it entered the ingress parser. When the clone operation is invoked at the end of egress pipeline, the cloned packet is a copy of the modified packet after egress processing, as output by the egress deparser. In both cases, the cloned packet is submitted to the egress pipeline for further processing.

Logically, PRE implements the mechanics of copying a packet. The metadata fields that control cloning are those whose names begin with `clone` in types `psa_ingress_output_metadata_t` and `psa_egress_output_metadata_t`.

```
bool                    clone;
CloneSessionId_t        clone_session_id;
```

The `clone` flag specifies whether a packet should be cloned. If true, then a cloned packet should be generated at the end of the pipeline. The `clone_session_id` specifies one of several possible clone sessions that the control plane may configure in the PRE. For each clone session, the control plane may configure the following values that should be associated with packets cloned using that session.

```
PortId_t         egress_port;
CloneSessionId_t class_of_service;
```

```
bool             truncate;
PacketLength_t   packet_length_bytes;  /// only used if truncate is true
```

The `egress_port` may be any port that can be used for normal unicast packets, i.e. any normal port, `PORT_CPU`, or `PORT_RECIRCULATE`. For the latter two values, the cloned packet will be sent to the CPU, or recirculated at the end of egress processing, as a normal unicast packet would at the end of egress processing.

Truncation of cloned packets is supported as an optimization to reduce the bandwidth required to send the beginning of packets. This is sometimes useful in sending packet headers to the control plane, or some kinds of data collection system for traffic monitoring. Here by "headers" we simply mean "some number of bytes from the beginning of the packet", not headers as defined and parsed in your P4 program.

If `truncate` is false for a clone session, then no truncation is performed for packets cloned using that session.

Otherwise, packets are truncated to contain at most the first `packet_length_bytes` bytes of the packet, with any additional bytes removed. Truncating a packet has no effect on any metadata that is carried along with it, and the size of that metadata is not counted as part of the `packet_length_bytes` quantity. Any truncation is based completely upon the length of the packet as passed to the type `packet_in` parameter to the ingress parser (for ingress to egress clones), or as sent out as the type `packet_out` parameter from the egress deparser (for egress to egress clones).

PSA implementations are allowed to support only a restricted set of possible values for `packet_length_bytes`, e.g. an implementation might choose only to support values that are multiples of 32 bytes.

Since it is an expected common case to clone packets to the CPU, every PSA implementation begins with a clone session `PSA_CLONE_SESSION_TO_CPU` initialized with `egress_port = PORT_CPU` and `class_of_service = 0`.

6.8.1. Clone Examples

The partial program below demonstrates how to clone a packet.

```
header clone_i2e_metadata_t {
    bit<8> custom_tag;
    EthernetAddress srcAddr;
}
control ingress(inout headers hdr,
                inout metadata user_meta,
                in  psa_ingress_input_metadata_t  istd,
                inout psa_ingress_output_metadata_t ostd)
{
    action do_clone (CloneSessionId_t session_id) {
        ostd.clone = true;
        ostd.clone_session_id = session_id;
        user_meta.custom_clone_id = 1;
    }
    table t {
        key = {
            user_meta.fwd_metadata.outport : exact;
        }
        actions = { do_clone; }
    }

    apply {
        t.apply();
```

```
    }
}
control IngressDeparserImpl(
    packet_out packet,
    out clone_i2e_metadata_t clone_i2e_meta,
    out empty_metadata_t resubmit_meta,
    out metadata normal_meta,
    inout headers hdr,
    in metadata meta,
    in psa_ingress_output_metadata_t istd)
{
    DeparserImpl() common_deparser;
    apply {
        // Assignments to the out parameter clone_i2e_meta must be
        // guarded by this if condition:
        if (psa_clone_i2e(istd)) {
            clone_i2e_meta.custom_tag = (bit<8>) meta.custom_clone_id;
            if (meta.custom_clone_id == 1) {
                clone_i2e_meta.srcAddr = hdr.ethernet.srcAddr;
            }
        }
        common_deparser.apply(packet, hdr);
    }
}
```

6.9. Packet Resubmission

Packet resubmission is a mechanism to repeat ingress processing on a packet.

Packet resubmission happens at the end of the ingress pipeline. When a packet is resubmitted, the packet finishes the ingress pipeline processing and re-enters the ingress parser without being deparsed. In other words, the resubmitted packet has the same header and payload as the original packet. The `ingress_port` of the resubmitted packet is the same as the original packet. The `packet_path` of the resubmitted packet is changed to `RESUBMIT`.

The ingress parser distinguishes the resubmitted packet from the original packet with the `packet_path` field in `ingress_parser_intrinsic_metadata_t`. The ingress parser can choose a different algorithm to parse the resubmitted packet. Similarly, the ingress pipeline can choose to process the resubmitted packet with different actions as opposed to the ones used to process the original packet. Further, if a target permits the same packet to be resubmitted multiple times, the user program can distinguish the packet resubmitted the first time, or second time, by the extra metadata associated with the packet. Note the maximum number of packet resubmission for a single packet is target-dependent. See section 3.

PSA specifies that the resubmit operation can only be used in the ingress pipeline. The egress pipeline cannot resubmit packets. As described in Section 3, there is no mandated mechanism in PSA to prevent a single received packet from creating packets that continue to recirculate, resubmit, or clone from egress to egress indefinitely. However, targets may impose limits on the number of resubmissions, recirculations, or clones.

One use case of packet resubmission is to increase the capacity and flexibility of the packet processing pipeline. For example, because the same packet is processed by the ingress pipeline multiple times, it effectively increase the amount of operations on the packet by N folds, where N is the number of times the packet is resubmitted.

Another use case is to deploy multiple packet processing algorithms on the same packet. For example, the original packet can be parsed and resubmitted in the first pass with additional metadata

to select one of the algorithms. Then, the resubmitted packet can be parsed, modified and deparsed using the selected algorithm.

To facilitate communication from the ingress processing pass that caused a resubmit to occur, to the next ingress processing pass after the resubmit has happened, the resubmission mechanism supports attaching optional metadata with the resubmitted packet. The metadata is generated during the pass through the ingress pipeline that chooses the resubmit operation, and used in the next pass.

A PSA implementation provides a configuration bit `resubmit` to the PRE to enable the resubmission mechanism. If true, the original packet is resubmitted with the optional resubmit metadata. If false, the resubmission mechanism is disabled and no assignments to `resubmit_meta` should be performed.

6.10. Packet Recirculation

Packet recirculation is a mechanism to repeat ingress processing on a packet, after it has completed egress processing. Unlike a resubmit, where the resubmitted packet contents are identical to the packet that arrived at the ingress parser, a recirculated packet may have different headers than the packet had before recirculation. This could be useful in implementing features such as multiple levels of tunnel encapsulation or decapsulation.

Whether a packet is recirculated must be chosen during ingress processing, by sending the packet to port `PORT_RECIRCULATE`. Packet recirculation happens at the end of the egress pipeline. When a packet is sent to the recirculate port, the packet finishes egress processing, including the egress deparser, and then re-enters the ingress parser. The `ingress_port` of the recirculated packet is set to `PORT_RECIRCULATE`. The `packet_path` of the recirculated packet is set to `RECIRCULATE`.

Similar to packet resubmission, packet recirculation also supports attaching optional metadata with the recirculated packet. The metadata is generated during egress processing, and filled in by assigning a value to the `out` parameter `recirculate_meta` of the egress deparser. The metadata is available to the ingress parser after the packet is recirculated.

7. PSA Externs

7.1. Restrictions on where externs may be used

All instantiations in a $P4_{16}$ program occur at compile time, and can be arranged in a tree structure we will call the instantiation tree. The root of the tree `T` represents the top level of the program. Its children are the node for the package `PSA_Switch` described in Section 5, and any externs instantiated at the top level of the program. The children of the `PSA_Switch` node are the packages and externs passed as parameters to the `PSA_Switch` instantiation. See Figure 3 for a drawing of the smallest instantiation tree possible for a P4 program written for PSA.
If any of those parsers or controls instantiate other parsers, controls, and/or externs, the instantiation tree contains child nodes for them, continuing until the instantiation tree is complete.

For every instance whose node is a descendant of the `Ingress` node in this tree, call it an `Ingress` instance. Similarly for the other ingress and egress parsers and controls. All other instances are top level instances.

A PSA implementation is allowed to reject programs that instantiate externs, or attempt to call their methods, from anywhere other than the places mentioned in Table 5.

For example, `Counter` being restricted to "Ingress, Egress" means that every `Counter` instance must be instantiated within either the `Ingress` control block or the `Egress` control block, or be a descendant of one of those nodes in the instantiation tree. If a `Counter` instance is instantiated in Ingress, for example, then it cannot be referenced, and thus its methods cannot be called, from any control block except `Ingress` or one of its descendants in the tree.

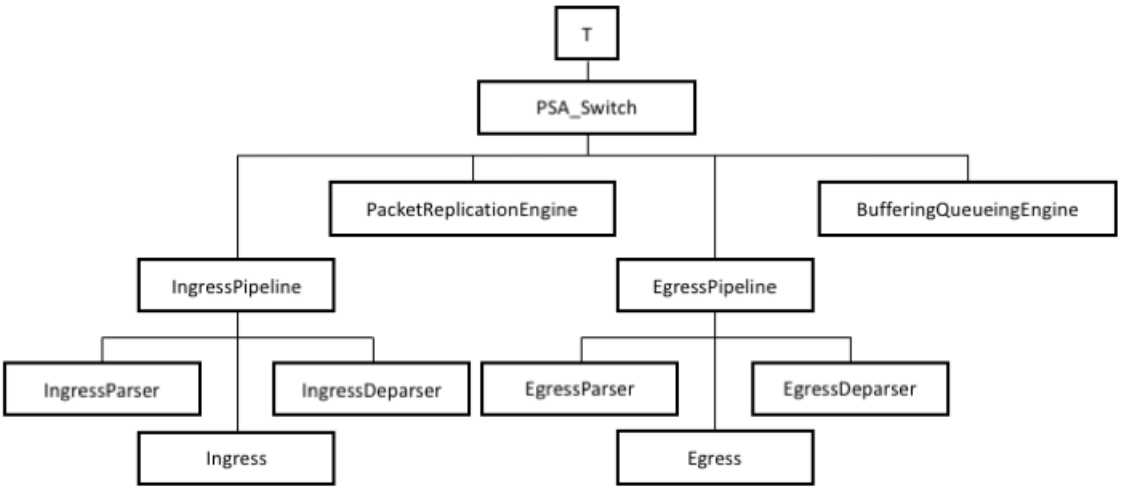

Figure 3. Minimal PSA instantiation tree

PSA implementations need not support instantiating these externs at the top level. PSA implementations are allowed to accept programs that use these externs in other places, but they need not. Thus P4 programmers wishing to maximize the portability of their programs should restrict their use of these externs to the places indicated in the table.

`emit` method calls for the type `packet_out` are restricted to be within deparser control blocks in PSA, because those are the only places where an instance of type `packet_out` is visible. Similarly all methods for type `packet_in`, e.g. `extract` and `advance`, are restricted to be within parsers in PSA programs. $P4_{16}$ restricts all `verify` method calls to be within parsers for all $P4_{16}$ programs, regardless of whether they are for the PSA.

Rationale:

- It is expected that the highest performance PSA implementations will not be able to update the same extern instance from both `Ingress` and Egress, nor from more than one of the parsers or controls defined in the PSA architecture.
- In a multi-pipeline device, there are effectively multiple instantiations of the ingress pipeline and of the egress pipeline. The primary motivation to create a multi-pipeline device is the practical difficulty in allowing the same stateful object (e.g. table, counter, etc.) to be accessed at a packet rate higher than that of a single pipeline. Thus each stateful object should be accessed from only a single pipeline on such a device.

7.2. PSA Table Properties

Table 6 lists all P4 table properties defined by PSA that are not included in the base $P4_{16}$ language specification.

A PSA implementation need not support both of a `psa_implementation` and `psa_direct_counters` property on the same table.

Similarly, a PSA implementation need not support both of a `psa_implementation` and `psa_direct_meters` property on the same table.

A PSA implementation must implement tables that have both a `psa_direct_counters` and `psa_direct_meters` property.

Extern type	Where it may be instantiated and called from
`ActionProfile`	Ingress, Egress
`ActionSelector`	Ingress, Egress
`Checksum`	IngressParser, EgressParser, IngressDeparser, EgressDeparser
`Counter`	Ingress, Egress
`Digest`	Ingress, Egress
`DirectCounter`	Ingress, Egress
`DirectMeter`	Ingress, Egress
`Hash`	Ingress, Egress
`InternetChecksum`	IngressParser, EgressParser, IngressDeparser, EgressDeparser
`Meter`	Ingress, Egress
`Random`	Ingress, Egress
`Register`	Ingress, Egress
`ValueSet`	IngressParser, EgressParser

Table 5. Summary of controls that can instantiate and invoke externs.

Property name	Type	See also
`psa_direct_counters`	list of `DirectCounter` instance names	Section 7.7.3
`psa_direct_meters`	list of `DirectMeter` instance names	Section 7.8
`psa_implementation`	instance name of one `ActionProfile` or `ActionSelector`	Sections 7.11, 7.12

Table 6. Summary of PSA table properties.

7.3. Packet Replication Engine

The `PacketReplicationEngine` extern (abbreviated PRE) represents a part of the PSA pipeline that is not programmable via writing P4 code.

Even though the PRE can not be programmed using P4, it can be configured using control plane APIs, e.g. configuring multicast groups and clone sessions. For every packet, your P4 program will typically assign values to intrinsic metadata in structs such as those of type `psa_ingress_output_metadata_t` and `psa_egress_output_metadata_t`, which direct the operation of the PRE on that packet. The file `psa.p4` defines some actions to help set these metadata fields for some common use cases, described in sections 6.3 and 6.6.

The PRE extern must be instantiated exactly once, in the `PSA_Switch` package instantiation. See near the end of Section 5 for the package definitions from `psa.p4`. See below for an example of instantiating these packages, including the instantiation of one instance of `PacketReplicationEngine` and one of `BufferingQueueingEngine` in the `PSA_Switch` package instantiation.

```
IngressPipeline(IngressParserImpl(),
                ingress(),
                IngressDeparserImpl()) ip;

EgressPipeline(EgressParserImpl(),
               egress(),
               EgressDeparserImpl()) ep;

PSA_Switch(ip, PacketReplicationEngine(), ep, BufferingQueueingEngine()) main;
```

7.4. Buffering Queuing Engine

The `BufferingQueueingEngine` extern (abbreviated BQE) represents another part of the PSA pipeline, after egress, that is not programmable via writing P4 code.

Even though the BQE can not be programmed using P4, it can be configured both directly using control plane APIs and by setting intrinsic metadata.

The BQE extern must be instantiated exactly once, as the PRE must. See Section 7.3 for additional discussion and example code.

7.5. Hashes

Supported hash algorithms:

```
enum HashAlgorithm_t {
  IDENTITY,
  CRC32,
  CRC32_CUSTOM,
  CRC16,
  CRC16_CUSTOM,
  ONES_COMPLEMENT16,  /// One's complement 16-bit sum used for IPv4 headers,
                      /// TCP, and UDP.
  TARGET_DEFAULT      /// target implementation defined
}
```

7.5.1. Hash function

Example usage:

```
parser P() {
  Hash<bit<16>>(HashAlgorithm_t.CRC16) h;
  bit<16> hash_value = h.get_hash(buffer);
}
```

Parameters:

- `algo` – The algorithm to use for computation (see 7.5).
- `O` – The type of the return value of the hash.

```
extern Hash<O> {
  /// Constructor
  Hash(HashAlgorithm_t algo);

  /// Compute the hash for data.
  /// @param data The data over which to calculate the hash.
  /// @return The hash value.
  O get_hash<D>(in D data);

  /// Compute the hash for data, with modulo by max, then add base.
  /// @param base Minimum return value.
  /// @param data The data over which to calculate the hash.
  /// @param max The hash value is divided by max to get modulo.
  ///        An implementation may limit the largest value supported,
  ///        e.g. to a value like 32, or 256.
  /// @return (base + (h % max)) where h is the hash value.
  O get_hash<T, D>(in T base, in D data, in T max);
}
```

7.6. Checksums

PSA provides checksum functions compute an integer on the stream of bytes in packet headers. Checksums are often used as an integrity check to detect corrupted or otherwise malformed packets.

7.6.1. Basic checksum

The basic checksum extern provided in PSA supports arbitrary hash algorithms.

Parameters:

- `W` – The width of the checksum

```
extern Checksum<W> {
  /// Constructor
  Checksum(HashAlgorithm_t hash);

  /// Reset internal state and prepare unit for computation
  void clear();

  /// Add data to checksum
  void update<T>(in T data);

  /// Get checksum for data added (and not removed) since last clear
  W    get();
}
```

7.6.2. Incremental checksum

PSA also provides an incremental checksum that comes equipped with an additional `subtract` method that can be used to remove data previously added. The checksum is computed using the `ONES_COMPLEMENT16` hash algorithm used with protocols such as IPv4, TCP, and UDP – see IETF RFC 1624 and section B for details.

```
// Checksum based on ‘ONES_COMPLEMENT16‘ algorithm used in IPv4, TCP, and UDP.
// Supports incremental updating via ‘remove‘ method.
// See IETF RFC 1624.
extern InternetChecksum {
  /// Constructor
  InternetChecksum();

  /// Reset internal state and prepare unit for computation.  Every
  /// instance of an InternetChecksum object is automatically
  /// initialized as if clear() had been called on it, once for each
  /// time the parser or control it is instantiated within is
  /// executed.  All state maintained by it is independent per packet.
  void clear();

  /// Add data to checksum.  data must be a multiple of 16 bits long.
  void add<T>(in T data);

  /// Subtract data from existing checksum.  data must be a multiple of
  /// 16 bits long.
  void subtract<T>(in T data);

  /// Get checksum for data added (and not removed) since last clear
```

```
  bit<16> get();

  /// Get current state of checksum computation.  The return value is
  /// only intended to be used for a future call to the set_state
  /// method.
  bit<16> get_state();

  /// Restore the state of the InternetChecksum instance to one
  /// returned from an earlier call to the get_state method.  This
  /// state could have been returned from the same instance of the
  /// InternetChecksum extern, or a different one.
  void set_state(bit<16> checksum_state);
}
```

7.6.3. InternetChecksum examples

The partial program below demonstrates one way to use the `InternetChecksum` extern to verify whether the checksum field in a parsed IPv4 header is correct, and set a parser error if it is wrong. It also demonstrates checking for parser errors in the `Ingress` control block, dropping the packet if any errors occurred during parsing. PSA programs may choose to handle packets with parser errors in other ways than shown in this example – it is up to the P4 program author to choose and write the desired behavior.

Neither $P4_{16}$ nor the PSA provide any special mechanisms to record the location within a packet that a parser error occurred. A P4 program author can choose to record such location information explicitly. For example, one may define metadata fields specifically for that purpose – e.g. to hold an encoded value representing the last parser state reached, or the number of bytes extracted so far – and then assign values to those fields within the parser state code.

```
// Define additional error values, one of them for packets with
// incorrect IPv4 header checksums.
error {
    UnhandledIPv4Options,
    BadIPv4HeaderChecksum
}

typedef bit<32> PacketCounter_t;
typedef bit<8>  ErrorIndex_t;

const bit<9> NUM_ERRORS = 256;

parser IngressParserImpl(packet_in buffer,
                         out headers hdr,
                         inout metadata user_meta,
                         in psa_ingress_parser_input_metadata_t istd,
                         in empty_metadata_t resubmit_meta,
                         in empty_metadata_t recirculate_meta)
{
    InternetChecksum() ck;
    state start {
        buffer.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            0x0800: parse_ipv4;
            default: accept;
```

```
        }
    }
    state parse_ipv4 {
        buffer.extract(hdr.ipv4);
        // TBD: It would be good to enhance this example to
        // demonstrate checking of IPv4 header checksums for IPv4
        // headers with options, but this example does not handle such
        // packets.
        verify(hdr.ipv4.ihl == 5, error.UnhandledIPv4Options);
        ck.clear();
        ck.add({
            /* 16-bit word  0   */ hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
            /* 16-bit word  1   */ hdr.ipv4.totalLen,
            /* 16-bit word  2   */ hdr.ipv4.identification,
            /* 16-bit word  3   */ hdr.ipv4.flags, hdr.ipv4.fragOffset,
            /* 16-bit word  4   */ hdr.ipv4.ttl, hdr.ipv4.protocol,
            /* 16-bit word  5 skip hdr.ipv4.hdrChecksum, */
            /* 16-bit words 6-7 */ hdr.ipv4.srcAddr,
            /* 16-bit words 8-9 */ hdr.ipv4.dstAddr
            });
        // The verify statement below will cause the parser to enter
        // the reject state, and thus terminate parsing immediately,
        // if the IPv4 header checksum is wrong.  It will also record
        // the error error.BadIPv4HeaderChecksum, which will be
        // available in a metadata field in the ingress control block.
        verify(ck.get() == hdr.ipv4.hdrChecksum,
               error.BadIPv4HeaderChecksum);
        transition select(hdr.ipv4.protocol) {
            6: parse_tcp;
            default: accept;
        }
    }
    state parse_tcp {
        buffer.extract(hdr.tcp);
        transition accept;
    }
}

control ingress(inout headers hdr,
                inout metadata user_meta,
                in    psa_ingress_input_metadata_t  istd,
                inout psa_ingress_output_metadata_t ostd)
{
    // Table parser_error_count_and_convert below shows one way to
    // count the number of times each parser error was encountered.
    // Although it is not used in this example program, it also shows
    // how to convert the error value into a unique bit vector value
    // 'error_idx', which can be useful if you wish to put a bit
    // vector encoding of an error into a packet header, e.g. for a
    // packet sent to the control CPU.

    DirectCounter<PacketCounter_t>(CounterType_t.PACKETS) parser_error_counts;
```

```
    ErrorIndex_t error_idx;

    action set_error_idx (ErrorIndex_t idx) {
        error_idx = idx;
        parser_error_counts.count();
    }
    table parser_error_count_and_convert {
        key = {
            istd.parser_error : exact;
        }
        actions = {
            set_error_idx;
        }
        default_action = set_error_idx(0);
        const entries = {
            error.NoError               : set_error_idx(1);
            error.PacketTooShort        : set_error_idx(2);
            error.NoMatch               : set_error_idx(3);
            error.StackOutOfBounds      : set_error_idx(4);
            error.HeaderTooShort        : set_error_idx(5);
            error.ParserTimeout         : set_error_idx(6);
            error.BadIPv4HeaderChecksum : set_error_idx(7);
            error.UnhandledIPv4Options  : set_error_idx(8);
        }
        psa_direct_counters = { parser_error_counts };
    }
    apply {
        if (istd.parser_error != error.NoError) {
            // Example code showing how to count number of times each
            // kind of parser error was seen.
            parser_error_count_and_convert.apply();
            ingress_drop(ostd);
            exit;
        }
        // Do normal packet processing here.
    }
}
```

The partial program below demonstrates one way to use the `InternetChecksum` extern to calculate and then fill in a correct IPv4 header checksum in the deparser block. In this example, the checksum is calculated fresh, so the outgoing checksum will be correct regardless of what changes might have been made to the IPv4 header fields in the Ingress (or Egress) control block that precedes it.

```
control EgressDeparserImpl(packet_out packet,
                           out empty_metadata_t clone_e2e_meta,
                           out empty_metadata_t recirculate_meta,
                           inout headers hdr,
                           in metadata meta,
                           in psa_egress_output_metadata_t istd,
                           in psa_egress_deparser_input_metadata_t edstd)
{
    InternetChecksum() ck;
    apply {
        ck.clear();
```

```
        ck.add({
            /* 16-bit word  0   */ hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
            /* 16-bit word  1   */ hdr.ipv4.totalLen,
            /* 16-bit word  2   */ hdr.ipv4.identification,
            /* 16-bit word  3   */ hdr.ipv4.flags, hdr.ipv4.fragOffset,
            /* 16-bit word  4   */ hdr.ipv4.ttl, hdr.ipv4.protocol,
            /* 16-bit word  5 skip hdr.ipv4.hdrChecksum, */
            /* 16-bit words 6-7 */ hdr.ipv4.srcAddr,
            /* 16-bit words 8-9 */ hdr.ipv4.dstAddr
            });
        hdr.ipv4.hdrChecksum = ck.get();
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
    }
}
```

As a final example, we can use the `InternetChecksum` to compute an incremental checksum for the TCP header. Recall the TCP checksum is computed over the *entire* packet, including the payload. Because the packet payload is not available in P4, we assume that the TCP checksum on the original packet is correct, and update it incrementally by invoking `subtract` and then `add` on any fields that are modified by the program. For example, the `Ingress` control in the program below updates the IPv4 source address, recording the original source address in a metadata field:

```
control ingress(inout headers hdr,
                inout metadata user_meta,
                in    psa_ingress_input_metadata_t  istd,
                inout psa_ingress_output_metadata_t ostd) {
    action drop() {
      ingress_drop(ostd);
    }
    action forward(PortId_t port, bit<32> srcAddr) {
      user_meta.fwd_metadata.old_srcAddr = hdr.ipv4.srcAddr;
      hdr.ipv4.srcAddr = srcAddr;
      send_to_port(ostd, port);
    }
    table route {
        key = { hdr.ipv4.dstAddr : lpm; }
        actions = {
          forward;
          drop;
        }
    }
    apply {
        if(hdr.ipv4.isValid()) {
          route.apply();
        }
    }
}
```

The deparser first updates the IPv4 checksum as above, and then incrementally computes the TCP checksum.

```
control EgressDeparserImpl(packet_out packet,
                           out empty_metadata_t clone_e2e_meta,
```

```
                          out empty_metadata_t recirculate_meta,
                          inout headers hdr,
                          in metadata user_meta,
                          in psa_egress_output_metadata_t istd,
                          in psa_egress_deparser_input_metadata_t edstd)
{
    InternetChecksum() ck;
    apply {
        // Update IPv4 checksum
        // This clear() call can be removed without affecting
        // behavior, as an InternetCheckum instance is automatically
        // cleared for each packet.
        ck.clear();
        ck.add({
            /* 16-bit word  0   */ hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
            /* 16-bit word  1   */ hdr.ipv4.totalLen,
            /* 16-bit word  2   */ hdr.ipv4.identification,
            /* 16-bit word  3   */ hdr.ipv4.flags, hdr.ipv4.fragOffset,
            /* 16-bit word  4   */ hdr.ipv4.ttl, hdr.ipv4.protocol,
            /* 16-bit word  5 skip hdr.ipv4.hdrChecksum, */
            /* 16-bit words 6-7 */ hdr.ipv4.srcAddr,
            /* 16-bit words 8-9 */ hdr.ipv4.dstAddr
            });
        hdr.ipv4.hdrChecksum = ck.get();
        // Update TCP checksum
        // This clear() call is necessary for correct behavior, since
        // the same instance 'ck' is reused from above for the same
        // packet.  If a second InternetChecksum instance other than
        // 'ck' were used below instead, this clear() call would be
        // unnecessary.
        ck.clear();
        // Subtract the original TCP checksum
        ck.subtract(hdr.tcp.checksum);
        // Subtract the effect of the original IPv4 source address,
        // which is part of the TCP 'pseudo-header' for the purposes
        // of TCP checksum calculation (see RFC 793), then add the
        // effect of the new IPv4 source address.
        ck.subtract(user_meta.fwd_metadata.old_srcAddr);
        ck.add(hdr.ipv4.srcAddr);
        hdr.tcp.checksum = ck.get();
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
    }
}
```

7.7. Counters

Counters are a mechanism for keeping statistics. The control plane can read counter values. A P4 program cannot read counter values, only update them. If you wish to implement a feature involving sequence numbers in packets, for example, use Registers instead (Section 7.9).

Direct counters are counters associated with a particular P4 table, and are implemented by the extern `DirectCounter`. There are also indexed counters, which are implemented by the extern

`Counter`. The primary differences between direct counters and indexed counters are:

- Number of independently updatable counter values:
  - A single instantiation of a direct counter always contains as many independent counter values as the number of entries in the table with which it is associated.
  - You must specify the number of independent counter values for an indexed counter when instantiating it. This number of counters need not be the same as the size of any table.
- Where counter updates are allowed in the P4 program:
  - For a direct counter, you may only invoke its `count` method from inside the actions of the table with which it is associated, and this always updates the counter value associated with the matching table entry.
  - For an indexed counter, you may invoke its `count` method anywhere in the P4 program where extern object method invocations are permitted (e.g. inside actions, or directly inside a control's `apply` block), and every such invocation must specify the index of the counter value to be updated.

Counters are only intended to support packet counters and byte counters, or a combination of both called `PACKETS_AND_BYTES`. The byte counts are always increased by some measure of the packet length, where the packet length used might vary from one PSA implementation to another. For example, one implementation might use the Ethernet frame length, including the Ethernet header and FCS bytes, as the packet arrived on a physical port. Another might not include the FCS bytes in its definition of the packet length. Another might only include the Ethernet payload length. Each PSA implementation should document how it determines the packet length used for byte counter updates.

If you wish to keep counts of other quantities, or to have more precise control over the packet length used in a byte counter, you may use Registers to achieve that (Section 7.9).

7.7.1. Counter types

```
enum CounterType_t {
    PACKETS,
    BYTES,
    PACKETS_AND_BYTES
}
```

7.7.2. Counter

```
/// Indirect counter with n_counters independent counter values, where
/// every counter value has a data plane size specified by type W.

extern Counter<W, S> {
  Counter(bit<32> n_counters, CounterType_t type);
  void count(in S index);

  /*
  /// The control plane API uses 64-bit wide counter values.  It is
  /// not intended to represent the size of counters as they are
  /// stored in the data plane.  It is expected that control plane
  /// software will periodically read the data plane counter values,
  /// and accumulate them into larger counters that are large enough
  /// to avoid reaching their maximum values for a suitably long
```

```
  /// operational time.  A 64-bit byte counter increased at maximum
  /// line rate for a 100 gigabit port would take over 46 years to
  /// wrap.

  @ControlPlaneAPI
  {
    bit<64> read      (in S index);
    bit<64> sync_read (in S index);
    void set          (in S index, in bit<64> seed);
    void reset        (in S index);
    void start        (in S index);
    void stop         (in S index);
  }
  */
}
```

See section C for pseudocode of an example implementation of the Counter extern.

PSA implementations must not update any counter values if an indexed counter is updated with an index that is too large. It is recommended that they count such erroneous attempted updates, and record other information that can help an P4 programmer debug such errors.

7.7.3. Direct Counter

```
extern DirectCounter<W> {
  DirectCounter(CounterType_t type);
  void count();

  /*
  @ControlPlaneAPI
  {
    W    read<W>      (in TableEntry key);
    W    sync_read<W> (in TableEntry key);
    void set          (in W seed);
    void reset        (in TableEntry key);
    void start        (in TableEntry key);
    void stop         (in TableEntry key);
  }
  */
}
```

A `DirectCounter` instance must appear in the list of values of the `psa_direct_counters` table attribute for exactly one table. We call this table the `DirectCounter` instance's "owner". It is an error to call the `count` method for a `DirectCounter` instance anywhere except inside an action of its owner table.

The counter value updated by an invocation of `count` is always the one associated with the table entry that matched.

An action of an owner table need not have `count` method calls for all of the `DirectCounter` instances that the table owns. You must use an explicit `count()` method call on a `DirectCounter` to update it, otherwise its state will not change.

An example implementation for the `DirectCounter` extern is essentially the same as the one for `Counter`. Since there is no `index` parameter to the `count` method, there is no need to check for whether it is in range.

The rules here mean that an action that calls `count` on a `DirectCounter` instance may only

be an action of that instance's one owner table. If you want to have a single action `A` that can be invoked by multiple tables, you can still do so by having a unique action for each such table with a `DirectCounter`, where each such action in turn calls action `A`, in addition to any `count` invocations they have.

A `DirectCounter` instance must have a counter value associated with its owner table that is updated when there is a default action assigned to the table, and a search of the table results in a miss. If there is no default action assigned to the table, then there need not be any counter updated when a search of the table results in a miss.

By "a default action is assigned to a table", we mean that either the table has a `default_action` table property with an action assigned to it in the P4 program, or the control plane has made an explicit call to assign the table a default action. If neither of these is true, then there is no default action assigned to the table.

7.7.4. Example program using counters

The following partial P4 program demonstrates the instantiation and updating of `Counter` and `DirectCounter` externs.

```
typedef bit<48> ByteCounter_t;
typedef bit<32> PacketCounter_t;
typedef bit<80> PacketByteCounter_t;

const PortId_t NUM_PORTS = 512;

struct headers {
    ethernet_t       ethernet;
    ipv4_t           ipv4;
}

control ingress(inout headers hdr,
                inout metadata user_meta,
                in    psa_ingress_input_metadata_t  istd,
                inout psa_ingress_output_metadata_t ostd)
{
    Counter<ByteCounter_t, PortId_t>((bit<32>) NUM_PORTS, CounterType_t.BYTES)
        port_bytes_in;
    DirectCounter<PacketByteCounter_t>(CounterType_t.PACKETS_AND_BYTES)
        per_prefix_pkt_byte_count;

    action next_hop(PortId_t oport) {
        per_prefix_pkt_byte_count.count();
        send_to_port(ostd, oport);
    }
    action default_route_drop() {
        per_prefix_pkt_byte_count.count();
        ingress_drop(ostd);
    }
    table ipv4_da_lpm {
        key = { hdr.ipv4.dstAddr: lpm; }
        actions = {
            next_hop;
            default_route_drop;
        }
```

```
        default_action = default_route_drop;
        psa_direct_counters = {
            // table ipv4_da_lpm owns this DirectCounter instance
            per_prefix_pkt_byte_count
        };
    }
    apply {
        port_bytes_in.count(istd.ingress_port);
        if (hdr.ipv4.isValid()) {
            ipv4_da_lpm.apply();
        }
    }
}

control egress(inout headers hdr,
               inout metadata user_meta,
               in    psa_egress_input_metadata_t  istd,
               inout psa_egress_output_metadata_t ostd)
{
    Counter<ByteCounter_t, PortId_t>((bit<32>) NUM_PORTS, CounterType_t.BYTES)
        port_bytes_out;
    apply {
        // By doing these stats updates on egress, then because
        // multicast replication happens before egress processing,
        // this update will occur once for each copy made, which in
        // this example is intentional.
        port_bytes_out.count(istd.egress_port);
    }
}
```

7.8. Meters

Meters (RFC 2698) are a more complex mechanism for keeping statistics about packets, most often used for dropping or "marking" packets that exceed an average packet or bit rate. To mark a packet means to change one or more of its quality of service values in packet headers such as the 802.1Q PCP (priority code point) or DSCP (differentiated service code point) bits within the IPv4 or IPv6 type of service byte. The meters specified in the PSA are 3-color meters.

PSA meters do not require any particular drop or marking actions, nor do they automatically implement those behaviors for you. Meters keep enough state, and update their state during `execute()` method calls, in such a way that they return a `GREEN` (also known as conform), `YELLOW` (exceed), or `RED` (violate) result. See RFC 2698 for details on the conditions under which one of these three results is returned. The P4 program is responsible for examining that returned result, and making changes to packet forwarding behavior as a result. The value returned by an uninitialized meter shall be GREEN. This is in accordance with the P4 Runtime specification.

RFC 2698 describes "color aware" and "color blind" variations of meters. The `Meter` and `DirectMeter` externs implement both. The only difference is in which `execute` method you use when updating them. See the comments on the `extern` definitions below.

Similar to counters, there are two flavors of meters: indexed and direct. (Indexed) meters are addressed by index, while direct meters always update a meter state corresponding to the matched table entry or action, and from the control plane API are addressed using P4 Runtime table entry as key.

There are many other similarities between counters and meters, including:

- The number of independently updatable meter values.
- Where meter updates are allowed in a P4 program.
- For `BYTES` type meters, the packet length used in the update is determined by the PSA implementation, and can vary from one PSA implementation to another.

Further similarities between direct counters and direct meters include:

- `DirectMeter` `execute` method calls must be performed within actions invoked by the table that owns the `DirectMeter` instance. It is optional for such an action to call the `execute` method.
- There must be a meter state associated with a `DirectMeter` instance's owner table, that can be updated when the table result is a miss. As for a `DirectCounter`, this state only needs to exist if a default action is assigned to the table.

The table attribute to specify that a table owns a `DirectMeter` instance is `psa_direct_meters`. The value of this table attribute is a list of meter instances.

As for counters, if you call the `execute(idx)` method on an indexed meter and `idx` is at least the number of meter states, so `idx` is out of range, no meter state is updated. The `execute` call still returns a value of type `MeterColor_t`, but the value is undefined – programs that wish to have predictable behavior across implementations must not use the undefined value in a way that affects the output packet or other side effects. The example code below shows one way to achieve predictable behavior. Note that this undefined behavior cannot occur if the value of `n_meters` of an indexed meter is 2^W, and the type `S` used to construct the meter is `bit<W>`, since the index value could never be out of range.

```
#define METER1_SIZE 100
Meter<bit<7>>(METER1_SIZE, MeterType_t.BYTES) meter1;
bit<7> idx;
MeterColor_t color1;

// ... later ...

if (idx < METER1_SIZE) {
    color1 = meter1.execute(idx, MeterColor_t.GREEN);
} else {
    // If idx is out of range, use a default value for color1.  One
    // may also choose to store an error flag in some metadata field.
    color1 = MeterColor_t.RED;
}
```

Any implementation will have a finite range that can be specified for the Peak Burst Size and Committed Burst Size. An implementation should document the maximum burst sizes they support, and if the implementation internally truncates the values that the control plane requests to something more coarse than any number of bytes, that should also be documented. It is recommended that the maximum burst sizes be allowed as large as the number of bytes that can be transmitted across the implementation's maximum speed port in 100 milliseconds.

Implementations will also have finite ranges and precisions that they support for the Peak Information Rate and Committed Information Rate. An implementation should document the maximum rate it supports, as well as the precision it supports for implementing requested rates. It is recommended that the maximum rate supported be at least the rate of the implementation's fastest port, and that the actual implemented rate should always be within plus or minus 0.1% of the requested rate.

7.8.1. Meter types

```
enum MeterType_t {
    PACKETS,
    BYTES
}
```

7.8.2. Meter colors

```
enum MeterColor_t { RED, GREEN, YELLOW };
```

7.8.3. Meter

```
// Indexed meter with n_meters independent meter states.

extern Meter<S> {
  Meter(bit<32> n_meters, MeterType_t type);

  // Use this method call to perform a color aware meter update (see
  // RFC 2698). The color of the packet before the method call was
  // made is specified by the color parameter.
  MeterColor_t execute(in S index, in MeterColor_t color);

  // Use this method call to perform a color blind meter update (see
  // RFC 2698).  It may be implemented via a call to execute(index,
  // MeterColor_t.GREEN), which has the same behavior.
  MeterColor_t execute(in S index);

  /*
  @ControlPlaneAPI
  {
    reset(in MeterColor_t color);
    setParams(in S index, in MeterConfig config);
    getParams(in S index, out MeterConfig config);
  }
  */
}
```

7.8.4. Direct Meter

```
extern DirectMeter {
  DirectMeter(MeterType_t type);
  // See the corresponding methods for extern Meter.
  MeterColor_t execute(in MeterColor_t color);
  MeterColor_t execute();

  /*
  @ControlPlaneAPI
  {
    reset(in TableEntry entry, in MeterColor_t color);
    void setConfig(in TableEntry entry, in MeterConfig config);
    void getConfig(in TableEntry entry, out MeterConfig config);
```

```
  }
  */
}
```

7.9. Registers

Registers are stateful memories whose values can be read and written during packet forwarding under the control of the P4 program. They are similar to counters and meters in that their state can be modified as a result of processing packets, but they are far more general in the behavior they can implement.

Although you may not use register contents directly in table match keys, you may use the `read()` method call on the right-hand side of an assignment statement, which retrieves the current value of the register. You may copy the register value into metadata, and it is then available for matching in subsequent tables. The value returned by an uninitialized register is undefined. If a target implementation chooses to return some value, the PSA recommends the value should be 0.

A simple usage example might be to verify that a "first packet" was seen for a particular type of flow. A register cell would be allocated to the flow, initialized to "clear". When the protocol signaled a "first packet", the table would match on this value and update the flow's cell to "marked". Subsequent packets in the flow could would be mapped to the same cell; the current cell value would be stored in metadata for the packet and a subsequent table could check that the flow was marked as active.

```
extern Register<T, S> {
  Register(bit<32> size);
  T    read  (in S index);
  void write (in S index, in T value);

  /*
  @ControlPlaneAPI
  {
    T    read<T>      (in S index);
    void set          (in S index, in T seed);
    void reset        (in S index);
  }
  */
}
```

Another example using registers is given below. It implements a packet and byte counter, where the byte counter can be updated by a packet length specified in the P4 program, rather than one chosen by the PSA implementation.

```
const PortId_t NUM_PORTS = 512;

// It would be more convenient to use a struct type to represent the
// state of a combined packet and byte count, and many other compound
// values one might wish to store in a Register instance.  However,
// the latest p4test as of 2017-Aug-13 does not allow a struct type to
// be returned from a method call like Register.read().

#define PACKET_COUNT_WIDTH 32
#define BYTE_COUNT_WIDTH 48
//#define PACKET_BYTE_COUNT_WIDTH (PACKET_COUNT_WIDTH + BYTE_COUNT_WIDTH)
#define PACKET_BYTE_COUNT_WIDTH 80
```

```
#define PACKET_COUNT_RANGE (PACKET_BYTE_COUNT_WIDTH-1):BYTE_COUNT_WIDTH
#define BYTE_COUNT_RANGE (BYTE_COUNT_WIDTH-1):0

typedef bit<PACKET_BYTE_COUNT_WIDTH> PacketByteCountState_t;

action update_pkt_ip_byte_count (inout PacketByteCountState_t s,
                                 in bit<16> ip_length_bytes)
{
    s[PACKET_COUNT_RANGE] = s[PACKET_COUNT_RANGE] + 1;
    s[BYTE_COUNT_RANGE] = (s[BYTE_COUNT_RANGE] +
                           (bit<BYTE_COUNT_WIDTH>) ip_length_bytes);
}

control ingress(inout headers hdr,
                inout metadata user_meta,
                in    psa_ingress_input_metadata_t  istd,
                inout psa_ingress_output_metadata_t ostd)
{
    Register<PacketByteCountState_t, PortId_t>((bit<32>) NUM_PORTS)
        port_pkt_ip_bytes_in;

    apply {
        ostd.egress_port = 0;
        if (hdr.ipv4.isValid()) {
            @atomic {
                PacketByteCountState_t tmp;
                tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);
                update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);
                port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);
            }
        }
    }
}
```

Note the use of the `@atomic` annotation in the block enclosing the `read()` and `write()` method calls on the `Register` instance. It is expected to be common that register accesses will need the `@atomic` annotation around portions of your program in order to behave as you desire. As stated in the $P4_{16}$ specification, without the `@atomic` annotation in this example, an implementation is allowed to process two packets `P1` and `P2` in parallel, and perform the register access operations in this order:

```
    // Possible order of operations for the example program if the
    // @atomic annotation is _not_ used.

    tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);  // for packet P1
    tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);  // for packet P2

    // At this time, if P1 and P2 came from the same ingress_port,
    // each of their values of tmp are identical.

    update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);    // for packet P1
    update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);    // for packet P2

    port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);  // for packet P1
```

```
    port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);  // for packet P2
    // The write() from packet P1 is lost.
```

Since different implementations may have different upper limits on the complexity of code that they will accept within an `@atomic` block, we recommend you keep them as small as possible, subject to maintaining your desired correct behavior.

Individual counter and meter method calls need not be enclosed in `@atomic` blocks to be safe – they guarantee atomic behavior of their individual method calls, without losing any updates.

As for indexed counters and meters, access to an index of a register that is at least the size of the register is out of bounds. An out of bounds write has no effect on the state of the system. An out of bounds read returns an undefined value. See the example in Section 7.8 for one way to write code to guarantee avoiding this undefined behavior. Out of bounds register accesses are impossible for a register instance with type `S` declared as `bit<W>` and size 2^W entries.

7.10. Random

The `Random` extern provides generation of pseudo-random numbers in a specified range with a uniform distribution. If one wishes to generate numbers with a non-uniform distribution, you may do so by first generating a uniformly distributed random value, and then using appropriate table lookups and/or arithmetic on the resulting value to achieve the desired distribution.

An implementation is not required to produce cryptographically strong pseudo-random number generation. For example, a particularly inexpensive implementation might use a linear feedback shift register to generate values.

```
extern Random<T> {
  Random(T min, T max);
  T read();

  /*
  @ControlPlaneAPI
  {
    void reset();
    void setSeed(in T seed);
  }
  */
}
```

7.11. Action Profile

Action profiles are used as table implementation attributes.

Action profiles provide a mechanism to populate table entries with action specifications that have been defined outside the table entry specification. An action profile extern can be instantiated as a resource in the P4 program. A table that uses this action profile must specify its `psa_implementation` attribute as the action profile instance.

Figure 4 contrasts a direct table with a table that has an action profile implementation. A direct table, as seen in Figure 4 (a) contains the action specification in each table entry. In this example, the table has a match key consisting of an LPM on header field `h.f`. The action is to set the port. As we can see, entries t1 and t3 have the same action, i.e. to set the port to 1. Action profiles enable sharing an action across multiple entries by using a separate table as shown in Figure 4 (b).

A table with an action profile implementation has entries that point to a member reference instead of directly defining an action specification. A mapping from member references to action specifications is maintained in a separate table that is part of the action profile instance defined in the table `psa_implementation` attribute. When a table with an action profile implementation is

Table entry	Key (h.f: lpm)	Action spec.
t1	01001*	set_port(1)
t2	1100*	set_port(2)
t3	101*	set_port(1)

(a) Direct table.

Table entry	Key (h.f: lpm)	Member ref.
t1	01001*	m1
t2	1100*	m2
t3	101*	m1

Member ref.	Action spec.
m1	set_port(1)
m2	set_port(2)

(b) Indirect table with action profile implementation.

Figure 4. Action profiles in PSA

applied, the member reference is resolved and the corresponding action specification is applied to the packet.

Action profile members may only specify action types defined in the `actions` attribute of the implemented table. An action profile instance may be shared across multiple tables only if all such tables define the same set of actions in their `actions` attribute. Tables with an action profile implementation cannot define a default action. The default action for such tables is implicitly set to `NoAction`.

The control plane can add, modify or delete member entries for a given action profile instance. The controller-assigned member reference must be unique in the scope of the action profile instance. An action profile instance may hold at most `size` entries as defined in the constructor parameter. Table entries must specify the action using the controller-assigned reference for the desired member entry. Directly specifying the action as part of the table entry is not allowed for tables with an action profile implementation.

```
extern ActionProfile {
  /// Construct an action profile of 'size' entries
  ActionProfile(bit<32> size);

  /*
  @ControlPlaneAPI
  {
     entry_handle add_member    (action_ref, action_data);
     void         delete_member (entry_handle);
     entry_handle modify_member (entry_handle, action_ref, action_data);
  }
  */
}
```

7.11.1. Action Profile Example

The P4 control block `Ctrl` in the example below instantiates an action profile `ap` that can contain at most 128 member entries. Table `indirect` uses this instance by specifying the `psa_implementation` attribute. The control plane can add member entries to `ap`, where each member can specify either a `foo` or `NoAction` action. Table entries for `indirect` table must specify the action using the

controller-assigned member reference.

```
control Ctrl(inout H hdr, inout M meta) {

  action foo() { meta.foo = 1; }

  action_profile ap(32w128);

  table indirect {
    key = {hdr.ipv4.dst_address: exact;}
    actions = { foo; NoAction; }
    psa_implementation = ap;
  }

  apply {
    indirect.apply();
  }
}
```

7.12. Action Selector

Action selectors are used as table implementation attributes.

Action selectors implement yet another mechanism to populate table entries with action specifications that have been defined outside the table entry. They are more powerful than action profiles because they also provide the ability to dynamically select the action specification to apply upon matching a table entry. An action selector extern can be instantiated as a resource in the P4 program, similar to action profiles. Furthermore, a table that uses this action selector must specify its `psa_implementation` attribute as the action selector instance.

Table entry	Key (h.f: lpm)	Member/ Group ref.
t1	01001*	g1
t2	1100*	m2
t3	101*	g2

Group ref.	Members
g1	m1, m2
g2	m1
g3	m2

Member ref.	Action spec.
m1	set_port(1)
m2	set_port(2)

Figure 5. Action selectors in PSA

Figure 5 illustrates a table that has an action selector implementation. In this example, the table has a match key consisting of an LPM on header field `h.f`. A second match type `selector` is used to define the fields that are used to look up the action specification from the selector at runtime.

A table with an action action selector implementation consists of entries that point to either an action profile member reference or an action profile group reference. An action selector instance can be logically visualized as two tables as shown in Figure 5. The first table contains a mapping from group references to a set of member references. The second table contains a mapping from member references to action specifications.

When a packet matches a table entry at runtime, the controller-assigned reference of the action profile member or group is read. If the entry points to a member then the corresponding action specification is applied to the packet. However, if the entry points to a group, a dynamic selection algorithm is used to select a member from the group, and the action specification corresponding to that member is applied. The dynamic selection algorithm is specified as a parameter when instantiating the action selector.

Action selector members may only specify action types defined in the `actions` attribute of the implemented table. All actions in a group must be of the same type. The action parameters for actions in the same group are allowed to differ, and the action of different groups in a selector may be different. An action selector instance may be shared across multiple tables only if all such tables define the same set of actions in their `actions` attribute. Furthermore, the selector match fields for such tables must be identical and must be specified in the same order across all tables sharing the selector. Tables with an action selector implementation cannot define a default action. The default action for such tables is implicitly set to `NoAction`.

The dynamic selection algorithm requires a field list as an input for generating the index to a member entry in a group. This field list is created by using the match type `selector` when defining the table match key. The match fields of type `selector` are composed into a field list in the order they are specified. The composed field list is passed as an input to the action selector implementation. It is illegal to define a `selector` type match field if the table does not have an action selector implementation.

The control plane can add, modify or delete member and group entries for a given action selector instance. An action selector instance may hold at most `size` member entries as defined in the constructor parameter. The number of groups may be at most the size of the table that is implemented by the selector. Table entries must specify the action using a reference to the desired member or group entry. Directly specifying the action as part of the table entry is not allowed for tables with an action selector implementation.

```
extern ActionSelector {
  /// Construct an action selector of 'size' entries
  /// @param algo hash algorithm to select a member in a group
  /// @param size number of entries in the action selector
  /// @param outputWidth size of the key
  ActionSelector(HashAlgorithm_t algo, bit<32> size, bit<32> outputWidth);

  /*
  @ControlPlaneAPI
  {
     entry_handle add_member        (action_ref, action_data);
     void         delete_member     (entry_handle);
     entry_handle modify_member     (entry_handle, action_ref, action_data);
     group_handle create_group      ();
     void         delete_group      (group_handle);
     void         add_to_group      (group_handle, entry_handle);
     void         delete_from_group (group_handle, entry_handle);
  }
  */
}
```

7.12.1. Action Selector Example

The P4 control block `Ctrl` in the example below instantiates an action selector `as` that can contain at most 128 member entries. The action selector uses a crc16 algorithm with output width of 10 bits to select a member entry within a group.

Table `indirect_with_selection` uses this instance by specifying the `psa_implementation` table property as shown. The control plane can add member and group entries to `as`. Each member can specify either a `foo` or `NoAction` action. When programming the table entries, the control plane *does not* include the fields of match type `selector` in the match key. The selector match fields are instead used to compose a list that is passed to the action selector instance. In the example below, the list {hdr.ipv4.src_address, hdr.ipv4.protocol} is passed as input to the crc16 hash algorithm

used for dynamic member selection by action selector `as`.

```
control Ctrl(inout H hdr, inout M meta) {

  action foo() { meta.foo = 1; }

  action_selector as(HashAlgorithm.crc16, 32w128, 32w10);

  table indirect_with_selection {
    key = {
      hdr.ipv4.dst_address: exact;
      hdr.ipv4.src_address: selector;
      hdr.ipv4.protocol: selector;
    }
    actions = { foo; NoAction; }
    psa_implementation = as;
  }

  apply {
    indirect_with_selection.apply();
  }
}
```

7.13. Parser Value Sets

A parser value set is a named set of values that may be used during packet header parsing time to make decisions. You may use control plane API calls to add values to a set, and remove values from a set, at run time, much like P4 tables. Unlike tables, they may not have actions associated with them. They may only be used to determine whether a particular value is in the set, returning a Boolean value. That Boolean value can then be used in a `select` statement to control parsing (see examples below).

```
extern ValueSet<D> {
    ValueSet(int<32> size);
    bool is_member(in D data);

    /*
    @ControlPlaneAPI
    message ValueSetEntry {
        uint32 value_set_id = 1;
        // FieldMatch allows specification of exact, lpm, ternary, and
        // range matching on fields for tables, and these options are
        // permitted for the ValueSet extern as well.
        repeated FieldMatch match = 2;
    }

    // ValueSetEntry should be added to the 'message Entity'
    // definition, inside its 'oneof Entity' list of possibilities.
    */
}
```

The control plane API excerpt above is intended to be added as part of the P4 Runtime API[1].

[1] The P4 Runtime API, defined as a Google Protocol Buffer `.proto` file, can be found at https://github.com/p4lang/PI/blob/master/proto/p4/p4runtime.proto

The control plane API for a `ValueSet` is similar to that of a table, except only match fields may be specified, with no actions. This includes API calls that specify ternary or range matching, although for `ValueSet`s these do not require specifying any priority values, since the only result of a `ValueSet` `is_member` call is "in the set" or "not in the set".

If a PSA target can do so, it should implement control plane API calls involving ternary or range matching using ternary or range matching capabilities in the target, consuming the minimal table entries possible.

However, a PSA target is allowed to implement such control plane API calls by "expanding" them into as many exact match entries as needed to have the same behavior. For example, a control plane API call adding all values in the range 5 through 8 may be implemented as adding the four separate exact match values 5, 6, 7, and 8.

The parser definition below shows an example that uses two `ValueSet` instances called `tpid_types` and `trill_types`.

```
parser IngressParserImpl(packet_in buffer,
                         out headers parsed_hdr,
                         inout metadata user_meta,
                         in psa_ingress_parser_input_metadata_t istd,
                         in empty_metadata_t resubmit_meta,
                         in empty_metadata_t recirculate_meta)
{
    ValueSet<bit<16>>(4) tpid_types;
    ValueSet<bit<16>>(2) trill_types;
    state start {
        buffer.extract(parsed_hdr.ethernet);
        transition select(parsed_hdr.ethernet.etherType) {
            0x0800: parse_ipv4;
            0x86DD: parse_ipv6;
            default: dispatch_tpid_value_set;
        }
    }
    state dispatch_tpid_value_set {
        bool is_tpid = tpid_types.is_member(parsed_hdr.ethernet.etherType);
        transition select(is_tpid) {
            true: parse_vlan_tag;
            default: dispatch_trill_value_set;
        }
    }
    state dispatch_trill_value_set {
        bool is_trill = trill_types.is_member(parsed_hdr.ethernet.etherType);
        transition select(is_trill) {
            true: parse_trill;
            default: accept;
        }
    }
    state parse_vlan_tag {
        // extract VLAN 802.1Q header here
        transition accept;
    }
    state parse_trill {
        // extract TRILL header here
        transition accept;
    }
```

```
    state parse_ipv4 {
        transition accept;
    }
    state parse_ipv6 {
        transition accept;
    }
}
```

The second example (below) has the same parsing behavior as the example above, but combines the two parse states `dispatch_tpid_value_set` and `dispatch_trill_value_set` into one.

```
    state dispatch_tpid_value_set {
        bool is_tpid = tpid_types.is_member(parsed_hdr.ethernet.etherType);
        bool is_trill = trill_types.is_member(parsed_hdr.ethernet.etherType);
        transition select(is_tpid, is_trill) {
            (true,     _): parse_vlan_tag;
            (false, true): parse_trill;
            default: accept;
        }
    }
```

The third example (below) demonstrates one way to have a `ValueSet` that matches on multiple fields, by making the type `D` a `struct` containing multiple bit vectors.

```
struct CustomValueSet1_t {
    bit<16> etherType;
    bit<8>  partialMacAddress;
}

parser IngressParserImpl(packet_in buffer,
                         out headers parsed_hdr,
                         inout metadata user_meta,
                         in psa_ingress_parser_input_metadata_t istd,
                         in empty_metadata_t resubmit_meta,
                         in empty_metadata_t recirculate_meta)
{
    ValueSet<CustomValueSet1_t>(2) trill_types;

    state dispatch_tpid_value_set {
        bool is_trill =
            trill_types.is_member({parsed_hdr.ethernet.etherType,
                                   parsed_hdr.ethernet.dstAddr[7:0]});
        transition select(is_trill) {
            true: parse_vlan_tag;
            default: accept;
        }
    }

    // ... etc.
```

A PSA compliant implementation is not required to support any use of a `ValueSet` `is_member` method call return value, other than directly inside of a `select` expression. For example, a program fragment like the one shown below may be rejected, and thus P4 programmers striving for maximum portability should avoid writing such code.

```
    bool is_tpid = tpid_types.is_member(parsed_hdr.ethernet.etherType);
```

```
is_tpid = is_tpid && (parsed_hdr.ethernet.dstAddr[47:40] == 0xfe);
transition select(is_tpid) {
    // ...
```

7.14. Timestamps

A PSA implementation provides an `ingress_timestamp` value for every packet in the `Ingress` control block, as a field in the struct with type `psa_ingress_input_metadata_t`. This timestamp should be close to the time that the first bit of the packet arrived to the device, or alternately, to the time that the device began parsing the packet. This timestamp is *not* automatically included with the packet in the `Egress` control block. A P4 program wishing to use the value of `ingress_timestamp` in egress code must copy it to a user-defined metadata field that reaches egress.

A PSA implementation also provides an `egress_timestamp` value for every packet in the `Egress` control block, as a field of the struct with type `psa_egress_input_metadata_t`.

One expected use case for timestamps is to store them in tables or `Register` instances to implement checking for timeout events for protocols, where precision on the order of milliseconds is sufficient for most protocols.

Another expected use case is INT (In-band Network Telemetry[2]), where precision on the order of microseconds or smaller is necessary to measure queueing latencies that differ by those amounts. It takes only 0.74 microseconds to transmit a 9 Kbyte Ethernet jumbo frame on a 100 gigabit per second link.

For these applications, it is recommended that an implementation's timestamp increments at least once every microsecond. Incrementing once per clock cycle in an ASIC or FPGA implementation would be a reasonable choice. The timestamp should increment at a constant rate over time. For example, it should not be a simple count of clock cycles in a device that implements dynamic frequency scaling[3].

Timestamps are of type `Timestamp_t`, which is type `bit<W>` for a value of `W` defined by the implementation. Timestamps are expected to wrap around during the normal passage of time. It is recommended that an implementation pick a rate of advance and a bit width such that wrapping around occurs at most once every hour. Making the wrap time this long (or longer) makes timestamps more useful for several use cases.

- Checking for timeouts of protocol hello / keep-alive traffic that is on the order of seconds or minutes.
- If timestamps are placed into packets without converting them to other formats, then external data analysis systems using those timestamps will in many cases need to do so, e.g. to compare timestamps stored in packets by different PSA devices. These systems will need different formulas and/or parameters to perform this conversion for each wrap period, or to add extra external time references to the recorded data. The extra data required for accurate conversion is lower, and the likelihood of conversion mistakes is lower, if the timestamp values wrap less often.
- If timestamps are converted to other formats within a P4 program, it will need access to parameters that are likely to change every wrap time, e.g. at least a "base value" to add some calculated value to. A straightforward way to do this requires the control plane to update these values at least once or twice per timestamp wrap time.
- Programs that wish to use `(egress_timestamp - ingress_timestamp)` to calculate the queueing latency experienced by a packet need the wrap time to exceed the maximum queueing latency.

Examples of the number of bits required for wrap times of at least one hour:

[2] http://p4.org/p4/inband-network-telemetry
[3] https://en.wikipedia.org/wiki/Dynamic_frequency_scaling

- A 32-bit timestamp advancing by 1 per microsecond takes 1.19 hours to wrap.
- A 42-bit timestamp advancing by 1 per nanosecond takes 1.22 hours to wrap.

A PSA implementation is not required to implement time synchronization, e.g. via PTP[4] or NTP[5].

The control plane API excerpt below is intended to be added as part of the P4 Runtime API.

```
// The TimestampInfo and Timestamp messages should be added to the
// "oneof" inside of message "Entity".

// TimestampInfo is only intended to be read.  Attempts to update this
// entity have no effect, and should return an error status that the
// entity is read only.

message TimestampInfo {
  // The number of bits in the device's `Timestamp_t` type.
  uint32 size_in_bits = 1;
  // The timestamp value of this device increments
  // `increments_per_period` times every `period_in_seconds` seconds.
  uint64 increments_per_period = 2;
  uint64 period_in_seconds = 3;
}

// The timestamp value can be read or written.  Note that if there are
// already timestamp values stored in tables or `Register` instances,
// they will not be updated as a result of writing this timestamp
// value.  Writing the device timestamp is intended only for
// initialization and testing.

message Timestamp {
  bytes value = 1;
}
```

For every packet `P` that is processed by ingress and then egress, with the minimum possible latency in the packet buffer, it is guaranteed that the `egress_timestamp` value for that packet will be the same as, or slightly larger than, the `ingress_timestamp` value that the packet was assigned on ingress. By "slightly larger than", we mean that the difference `(egress_timestamp - ingress_timestamp)` should be a reasonably accurate estimate of this minimum possible latency through the packet buffer, perhaps truncated down to 0 if timestamps advance more slowly than this minimum latency.

Consider two packets such that at the same time (e.g. the same clock cycle), one is assigned its value of `ingress_timestamp` near the time it begins parsing, and the other is assigned its value of `egress_timestamp` near the time that it begins its egress processing. It is allowed that these timestamps differ by a few tens of nanoseconds (or by one "tick" of the timestamp, if one tick is larger than that time), due to practical difficulties in making them always equal.

Recall that the binary operators `+` and `-` on the `bit<W>` type in P4 are defined to perform wrap-around unsigned arithmetic. Thus even if a timestamp value wraps around from its maximum value back to 0, you can always calculate the number of ticks that have elapsed from timestamp $t1$ until timestamp $t2$ using the expression $(t2 - t1)$ (if more than 2^W ticks have elapsed, there will be aliasing of the result). For example, if timestamps were $W >= 4$ bits in size, $t1 = 2^W - 5$, and $t2 = 3$, then $(t2 - t1) = 8$.

It is sometimes useful to minimize storage costs by discarding some bits of a timestamp value in a P4 program for use cases that do not need the full wrap time or precision. For example, an

[4] https://en.wikipedia.org/wiki/Precision_Time_Protocol
[5] https://en.wikipedia.org/wiki/Network_Time_Protocol

application that only needs to detect protocol timeouts with an accuracy of 1 second can discard the least significant bits of a timestamp that change more often than every 1 second.

Another example is an application that needed full precision of the least significant bits of a timestamp, but the combination of the control plane and P4 program are designed to examine all entries of a `Register` array where these partial timestamps are stored more often than once every 5 seconds, to prevent wrapping. In that case, the P4 program could discard the most significant bits of the timestamp so that the remaining bits wrap every 8 seconds, and store those partial timestamps in the `Register` instance.

7.15. Packet Digest

Packet digest is a mechanism to perform a upcall to the control plane. A upcall is a way to send information from the lower layer of the software stack to an upper layer. In the case of packet digest, the information is sent from data-plane to control plane.

A PSA implementation should provide a digest mechanism. The digest contains two pieces of information: the receiver identifier and the digest message. The receiver identifier is an opaque id used by the control plane to route the digest message to the right receiver. The digest message can contain any from the data-plane.

In PSA, digest is performed at the end of the ingress pipeline in the deparser.

TBD: The digest extern provides a `pack` method to specify the content in the digest header.

The compiler decides the best serialization format to send the data object to the control plane.

On the control plane side, the control plane API calls the `unpack` method, and pass in a reference to the data object to be read.

The `unpack` method return 0 if succeed. The field names in the control plane API is derived from the field names in the data-plane `pack` method.

7.15.1. Control Plane

The control plane should provide a

```
extern Digest<T> {
  Digest(PortId_t receiver);         /// define a digest stream to receiver
  void pack(in T data);            /// emit data into the stream

  /*
  @ControlPlaneAPI
  {
  T data;                            /// If T is a list, control plane generates a struct.
  int unpack(T& data);               /// unpacked data is in T&, int return status code.
  }
  */
}
```

A. Appendix: Open Issues

As with any work in progress, we have a number of open issues that are under discussion in the working group. In addition to the TBDs in the document, there a number of larger issues that are summarized here:

A.1. Action Selectors

The size parameter in the action_selector instance that defines the maximum number of members in a selector. In some cases it might be useful to allow the controller to dynamically provision resources on the selector or to utilize different selector sizes on different targets, while using a common P4 program.

We also need to formalize the interaction of action profiles and action selectors with counters and meters.

A.2. How does PSA interact with multiple pipelines

As presented, the PSA describes the operation of a single packet processing pipeline. However, programmable devices typically support multiple pipelines. It is unlikely that high performance devices implement any form of coherency across pipelines, even though forwarding across pipelines may happen.

In the current specification, each extern has its own resources and they are constrained to a single pipeline, i.e., not shared across pipelines and not maintained coherent with any other externs even if attached to the same logical table. It is the responsibility of the programmer to maintain state across pipelines using control-plane operations, if so desired.

Likewise, for programming multiple pipelines, it is the responsibility of the vendor and of target dependent tools to specify how PSA programs are mapped to multiple pipelines. A simple implementation may use a copy of the PSA program on each pipeline, and thus keeping pipelines fully isolated. We will continue to discuss and refine our position.

A.3. PSA profiles

We are considering whether to specify different limits that a certain PSA implementation has to have in order for the implementation to be considered compliant. The main point of PSA is to enable a variety of devices, and thus limits may be artificial. On the other hand, for most interesting applications, it is necessary to support a minimum of functionality.

B. Appendix: Implementation of the InternetChecksum extern

Besides RFC 1461, RFC 1071 and RFC 1141 also contain useful tips on efficiently computing the Internet checksum, especially in software implementations.

Here we give reference implementations for the methods of the `InternetChecksum` extern, specified with the syntax and semantics of $P4_{16}$, with extensions of a `for` loop and a `return` statement for returning a value from a function.

The minimum internal state necessary for one instance of an `InternetChecksum` object is a 16-bit bit vector, here called `sum`.

```
// This is one way to perform a normal one's complement sum of two
// 16-bit values.
bit<16> ones_complement_sum(in bit<16> x, in bit<16> y) {
    bit<17> ret = (bit<17>) x + (bit<17>) y;
    if (ret[16:16] == 1) {
        ret = ret + 1;
    }
    return ret[15:0];
}

bit<16> sum;
```

```
void clear() {
    sum = 0;
}

// Restriction: data is a multiple of 16 bits long
void update<T>(in T data) {
    bit<16> d;
    for (each 16-bit aligned piece d of data) {
        sum = ones_complement_sum(sum, d);
    }
}

// Restriction: data is a multiple of 16 bits long
void remove<T>(in T data) {
    bit<16> d;
    for (each 16-bit aligned piece d of data) {
        // ~d is the negative of d in one's complement arithmetic.
        sum = ones_complement_sum(sum, ~d);
    }
}

// The Internet checksum is the one's complement _of_ the one's
// complement sum of the relevant parts of the packet.  The methods
// above calculate the one's complement sum of the parts in the
// variable 'sum'.  get() returns the bitwise negation of 'sum', which
// is the one's complement of 'sum'.

bit<16> get() {
    return ~sum;
}

bit<16> get_state() {
    return sum;
}

void set_state(bit<16> checksum_state) {
    sum = checksum_state;
}
```

C. Appendix: Example implementation of Counter extern

The example implementation below, in particular the function `next_counter_value`, is not intended to restrict PSA implementations. The storage format for `PACKETS_AND_BYTES` type counters demonstrated there is one example of how it could be done. Implementations are free to store state in other ways, as long as the control plane API returns the correct packet and byte count values.

Two common techniques for counter implementations in the data plane are:

- wrap around counters
- saturating counters, that 'stick' at their maximum possible value, without wrapping around.

This specification does not mandate any particular approach in the data plane. Implementations should strive to avoid losing information in counters. One common implementation technique is to implement an atomic "read and clear" operation in the data plane that can be invoked by the

control plane software. The control plane software invokes this operation frequently enough to prevent counters from ever wrapping or saturating, and adds the values read to larger counters in driver memory.

```
Counter(bit<32> n_counters, CounterType_t type) {
    this.num_counters = n_counters;
    this.counter_vals = new array of size n_counters, each element with type W;
    this.type = type;
    if (this.type == CounterType_t.PACKETS_AND_BYTES) {
        // Packet and byte counts share storage in the same counter
        // state.  Should we have a separate constructor with an
        // additional argument indicating how many of the bits to use
        // for the byte counter?
        W shift_amount = TBD;
        this.shifted_packet_count = ((W) 1) << shift_amount;
        this.packet_count_mask = (~((W) 0)) << shift_amount;
        this.byte_count_mask = ~this.packet_count_mask;
    }
}

W next_counter_value(W cur_value, CounterType_t type) {
    if (type == CounterType_t.PACKETS) {
        return (cur_value + 1);
    }
    // Exactly which packet bytes are included in packet_len is
    // implementation-specific.
    PacketLength_t packet_len = <packet length in bytes>;
    if (type == CounterType_t.BYTES) {
        return (cur_value + packet_len);
    }
    // type must be CounterType_t.PACKETS_AND_BYTES
    // In type W, the least significant bits contain the byte
    // count, and most significant bits contain the packet count.
    // This is merely one example storage format.  Implementations
    // are free to store packets_and_byte state in other ways, as
    // long as the control plane API returns the correct separate
    // packet and byte count values.
    W next_packet_count = ((cur_value + this.shifted_packet_count) &
                           this.packet_count_mask);
    W next_byte_count = (cur_value + packet_len) & this.byte_count_mask;
    return (next_packet_count | next_byte_count);
}

void count(in S index) {
    if (index < this.num_counters) {
        this.counter_vals[index] = next_counter_value(this.counter_vals[index],
                                                      this.type);
    } else {
        // No counter_vals updated if index is out of range.
        // See below for optional debug information to record.
    }
}
```

Optional debugging information that may be kept if an `index` value is out of range includes:

- Number of times this occurs.
- A FIFO of the first N out-of-range index values that occur, where N is implementation-defined (e.g. it might only be 1). Extra information to identify which `count()` method call in the P4 program had the out-of-range `index` value is also recommended.

D. Appendix: Rationale for design

D.1. Why egress processing?

Question: Why is it useful to have separate ingress vs. egress processing in a switch device?

There have been packet processing ASICs built that effectively only do 'ingress' processing, then go to a packet buffer with one or more queues, and then go out of the device, effectively being restricted to no or "empty" egress processing.

There are a few things that are trickier to do in such a device.

1. Last-nanosecond changes to the packet

If you want to measure the queuing latency through the device, and put a measurement of this quantity inside the packet somewhere, it is in general not possible to know the queueing latency before the packet is sent to the packet buffer. There are *special cases* where you can predict it, e.g. when there is a single FIFO queue feeding a constant bit rate output port, with nothing like Ethernet pause flow control.

But if you have variable bit rate links, e.g. because of things like Ethernet pause flow control, or Wi-Fi signal quality changes, or if you have multiple class-of-service queues with a scheduling policy between them like weighted fair queueing, then it is not possible to predict at the time the packet is enqueued, when it will be dequeued. The queueing latency depends upon unknown future events, such as whether Ethernet pause frames will arrive, or how many and what size of packets arriving in the near future will be put into which class of service queues for the same output port.

In such cases, having egress processing for taking the measurement, after it is known and easy to calculate as "dequeue time - enqueue time", allows the egress processing to modify the packet further.

2. Multicast efficiency and flexibility

It is possible in a PSA device to handle multicast by doing a recirculate plus clone operation for each of N copies to be made, but this reduces the processing capacity of ingress that is available to newly arriving packets, in particular newly arriving packets that you might consider more important to keep than the multicast packets.

By designing a packet buffer that can take a packet with a 'multicast group id', which the control plane configures to make copies to a selected set of output ports, it frees up the part of the system that performs ingress processing to accept new packets more quickly, and at a more predictable rate.

There could still be a challenge in designing the packet replication portion of the system not to fall behind when many multicast packets to be replicated to many output ports arrive close together in time, but it is fairly easy to separate the concerns of multicast from unicast packets. For example, a device implementer could prioritize unicast packets so that they are not slowed down if multicast replication is falling behind.

Once you have multicast designed in this way, there are still multicast use cases where one needs to process different copies of the packet differently. For example, the copy going out port 5 might need a VLAN tag of 7 placed in its header, whereas the copy going out port 2 might need a VLAN tag of 18 placed in its header. Similarly for multicast packets entering one of many flavors of tunnels, e.g. VXLAN, GRE, etc. By doing this per-copy modifications in egress processing, the packet replication logic can be kept very simple – just make identical copies of the packet as ingress finished with it, except for some kind of unique 'id' on each copy that egress processing can use to distinguish them.

D.2. No output port change during egress

Question: Why can't my P4 program change the output port during egress processing?

In a network device that has many input and output ports, packets can arrive at or near the same time on multiple input ports, all destined for the same output port.

Packet buffers are typically designed into such network devices, to store the packets that cannot be sent out immediately, absorbing this short term congestion.

For a given output port P, we now wish to retrieve packets from the packet buffer at a rate that is equal to the rate we will send them to port P, typically equal to the maximum bit rate that it is possible to send data out of port P.

Packet scheduling algorithms such as weighted fair queueing, and many others, have been developed that can determine which among a set of potentially many FIFO queues that a packet should be read from next, and sent out on the port.

These link scheduling algorithms are real time algorithms with very tight timing constraints. If they go too slow, the output port goes idle and its capacity is wasted. If they go too fast, we read packets from the packet buffer faster than they can be transmitted on the port, and we are back at the same problem we had originally – either drop some of the packets, or store them somewhere again until the port is ready to transmit them.

Such a scheduling algorithm that handles multiple output ports must know which output port all packets are destined to, before they are put into the packet buffer. If that target output port can be changed after the packet is read out, then we can simultaneously overload one output port while starving another.

That is why the `egress_port` of a packet must be selected during ingress processing, and egress processing is not allowed to change it.

These scheduling algorithms also need to know the size of each packet, i.e. the size as it will be when transmitted on the port.

It is possible in egress P4 code to drop a packet, or to change the size of the packet by adding or removing headers. Very likely P4-programmable network devices will have their scheduling algorithms run just slightly faster than the port to handle cases where many packets in a row are decreased in size during egress processing, and have tight control loops monitoring the size of packets leaving egress processing to make small adustments in the rate that the scheduling algorithm operates for each port. Either that, or they will just leave some fraction of the output port's capacity unused during times when all packet sizes are being decreased.

Certainly if there are long durations of time when egress decides to drop all packets to an output port, that port will go idle. The scheduling algorithm implementations are all built with a finite maximum packet scheduling rate.